PART IV

Composite Types

Gopher Level

Intermediate

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

PART IV

Composite Types

Arrays ✓

Collection
of
Elements
Indexable
Fixed Length

Slices

Collection
of
Elements
Indexable
Dynamic length

String Internals

Byte Slices
ASCII & Unicode
Encoding & Decoding

Maps

Collection
of
Indexable
Key-Value Pairs

Structs

Groups
different types of
variables together

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

PART IV

Composite Types

✓

Arrays

Collection
of
Elements
Indexable
Fixed Length

Slices

Collection
of
Elements
Indexable
Dynamic length

String Internals

Byte Slices
ASCII & Unicode
Encoding & Decoding

Maps

Collection
of
Indexable
Key-Value Pairs

Structs

Groups
different types of
variables together

A slice doesn't contain any elements on its own;
It's just a view into an array.

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICES

What you're going to learn in this section?

- ★ Slices vs Arrays
- ★ Growing — append()
- ★ Slicing — Slice Expressions *[low:high]*
- ★ Internals
  - ★ Backing Array
  - ★ Slice Header
  - ★ Capacity of a Slice — cap()
- ★ Full Slice Expressions *[low:high:cap]*
- ★ Preallocation — make()
- ★ Copying slices without a loop — copy()
- ★ Multi-Dimensional Slices

SLICES

You're going to build awesome projects again!

EMPTY FILE FINDER

Your first taste of file operations

```
$ go run main.go ~/files
Empty files in "~/files":
empty file 1
empty file 2
```

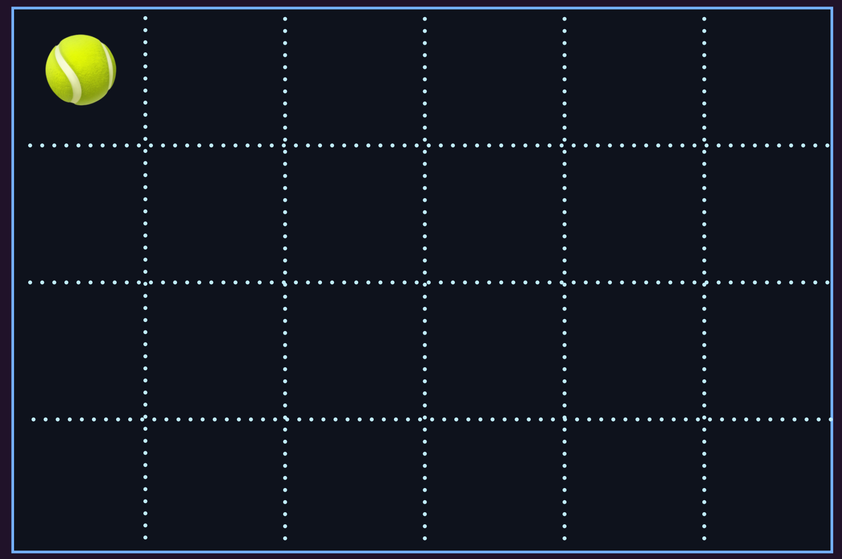
→ out.txt

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICES

You're going to build awesome projects again!

BOUNCING BALL



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICES vs ARRAYS

Let's talk about the differences between slices and arrays

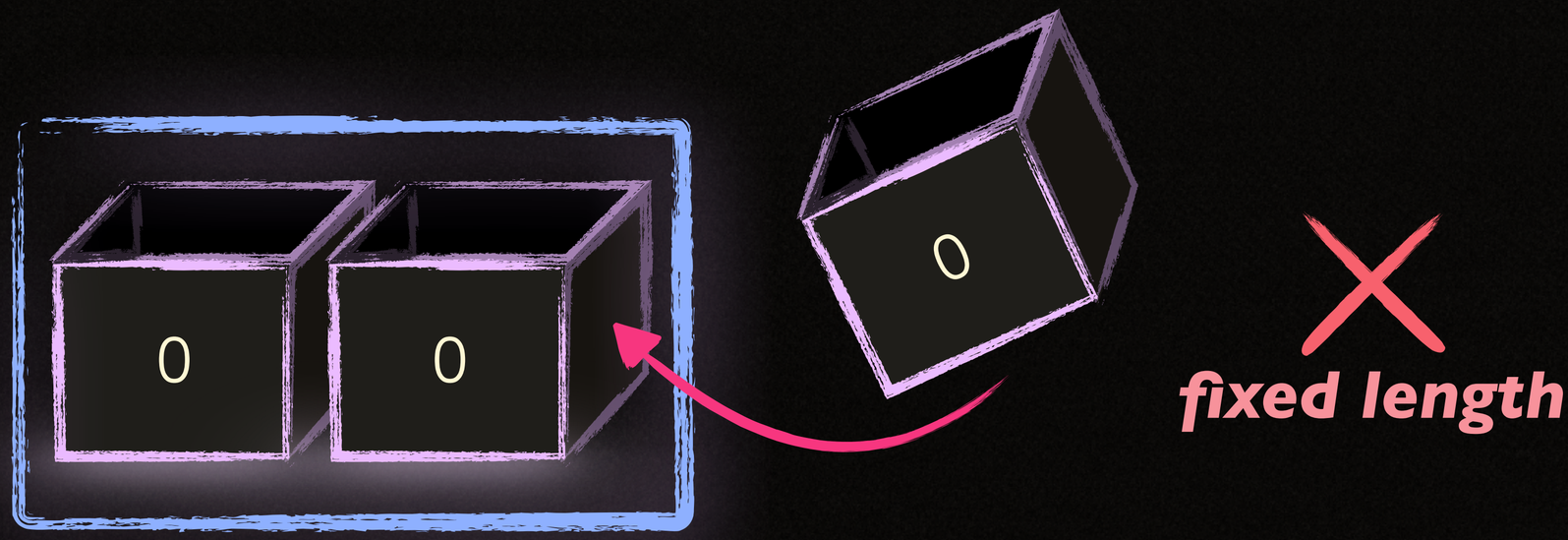
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

Why?

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

ARRAYS

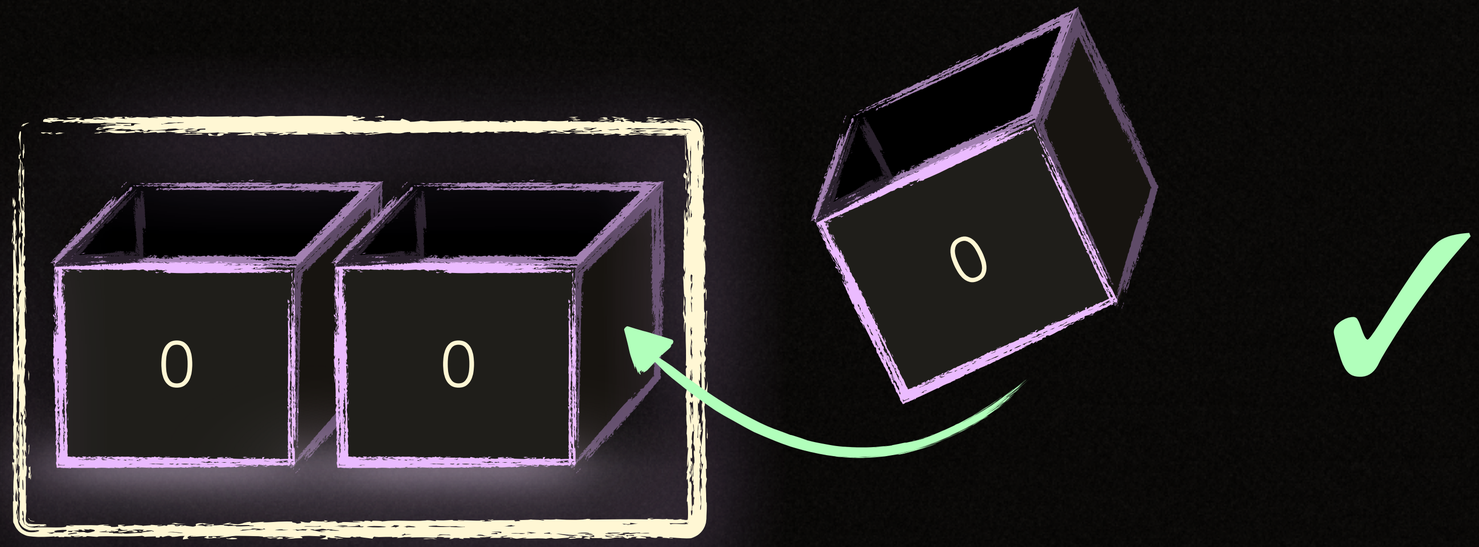
Arrays cannot grow or shrink



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICES

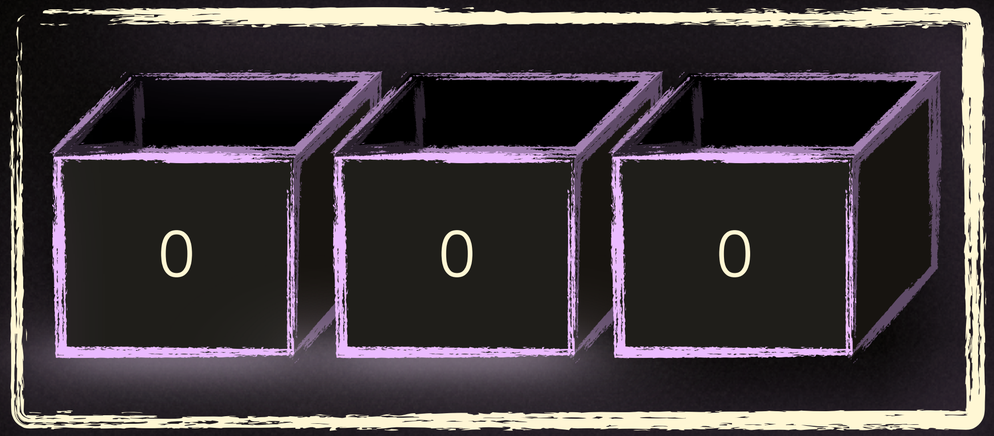
Slices can grow and shrink in runtime



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICES

Slices can grow and shrink in runtime



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

ARRAY

You cannot add or remove elements to/from an array in runtime

```
var nums [2]int
```

Its length is part of its type

- ✗ Its length belongs to compile-time
- ✗ So its length cannot change in runtime
- ✗ So it cannot grow or shrink

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE

You can add or remove elements to/from a slice in runtime

```
var nums []int
```

It doesn't have a fixed length at *compile-time*

✓ So it can grow and shrink in runtime

✓ Its length belongs to runtime

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

ARRAY'S ZERO VALUE

Go automatically sets uninitialized array elements to their Zero Values

```
var nums [5]int
```

nums is [0, 0, 0, 0, 0]

len(nums) is equal to 5

It always have the same number of elements

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE'S ZERO VALUE

Unlike an array, a slice's zero value is nil

```
var nums []int
```

nums is nil

len(nums) is equal to 0

It doesn't contain any elements yet

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

NIL SLICE

You cannot get and set non-existing elements from/to a nil slice

```
var nums []int

nums[0]  ⟶  ✗
nums[1]  ⟶  ✗
len(hello)  ⟶  0  ✓
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICES vs ARRAYS

Slices are dynamic — Arrays are fixed

SLICE

```
var nums []int
```

Its length is *not* part of its type

It can grow and shrink

Its zero value is nil

ARRAY

```
var nums [2]int
```

Its length is part of its type

It cannot grow or shrink

Its zero value is zero valued elements

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICES & ARRAYS

Common properties

SLICE

```
var nums []int
```

ARRAY

```
var nums [2]int
```

They can only contain the same type of elements

nums[0]

nums[1]

len(nums)

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

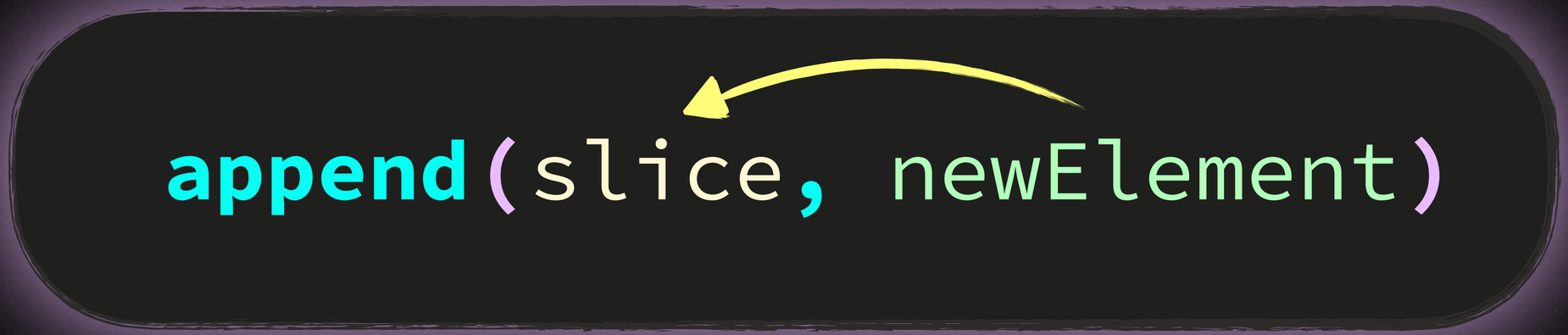
APPEND

Time to grow up!

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

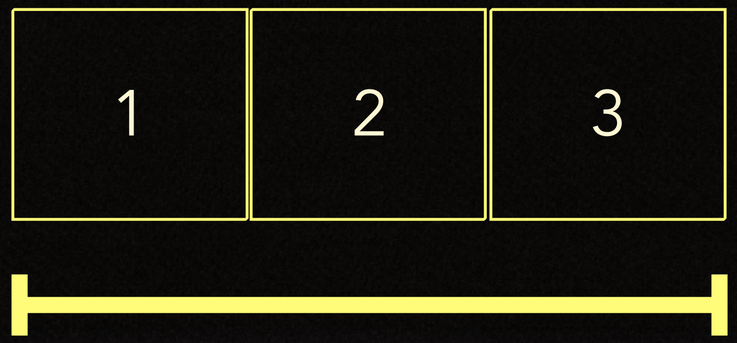
Appends a new element to a slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appends a new element to a slice



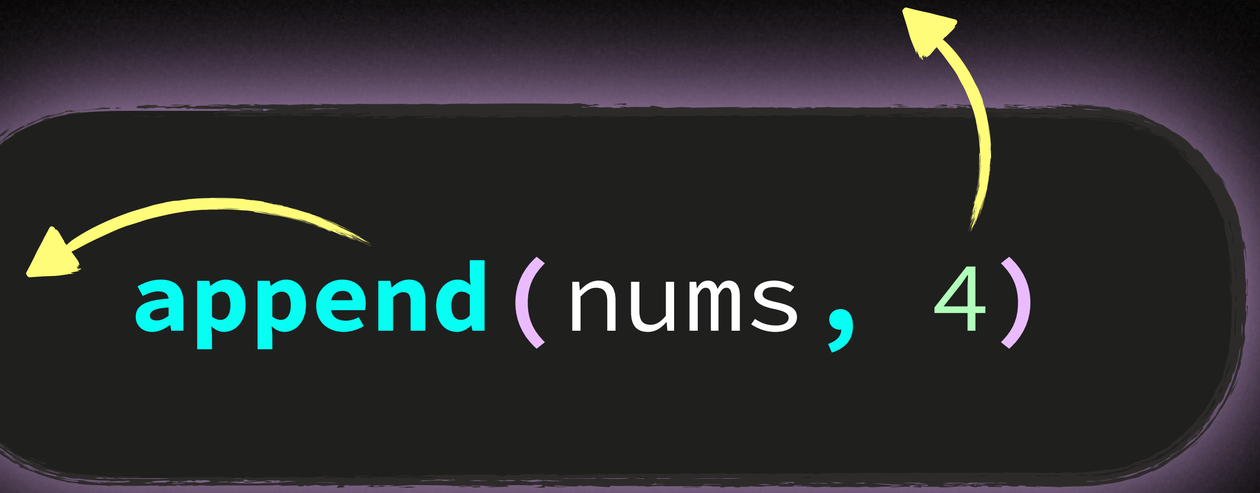

```
nums := []int{1, 2, 3}
append(nums, 4)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appends an element to a slice and returns a new slice

appends it to a new slice



append can't change the passed slice

Instead: It returns **a new slice**

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appends an element to a slice and returns a new slice

append returns a new slice

Here, I'm **saving** it **back** to the **nums slice**

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appends an element to a slice and returns a new slice

```
nums := []int{1, 2, 3}
nums = append(nums, 4)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appends an element to a slice and returns a new slice

```
nums := []int{1, 2, 3}
nums = append(nums, 4)
nums = append(nums, 9)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appends new elements to the back

```
nums := []int{1, 2, 3}
nums = append(nums, 4)
nums = append(nums, 9)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

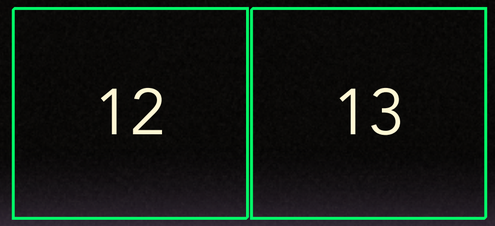
It can append multiple elements — append is a variadic function

```
nums := []int{1, 2, 3}
nums = append(nums, 4, 9)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Ellipsis allows you to append a slice to another slice




```
nums := []int{1, 2, 3}
tens := []int{12, 13}
nums = append(nums, tens...)
```

```
append(nums, 12, 13)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Let's cut that slice!

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

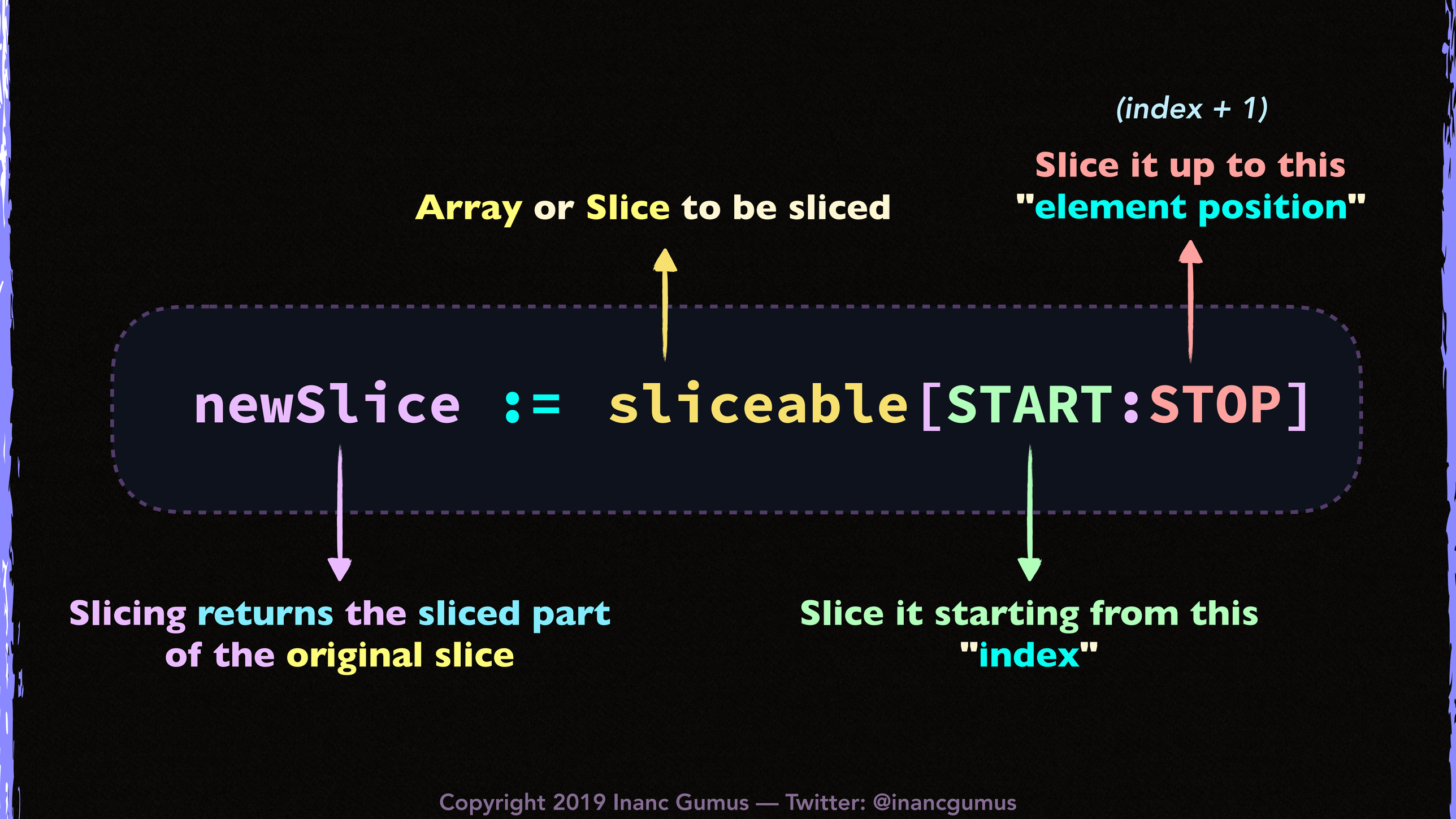
SLICEABLE = Any value that can be sliced

SLICE

ARRAY

STRING

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

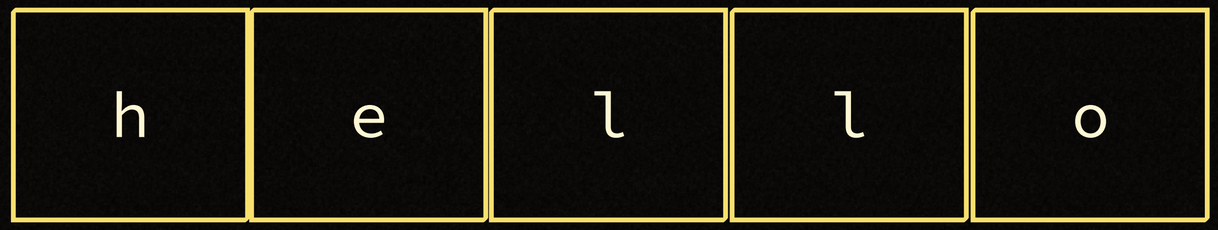

(index + 1)
Slice it up to this "element position"
Array or Slice to be sliced
newSlice := sliceable[START:STOP]
Slicing returns the sliced part of the original slice
Slice it starting from this "index"

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Slicing creates a new slice by cutting a sliceable

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

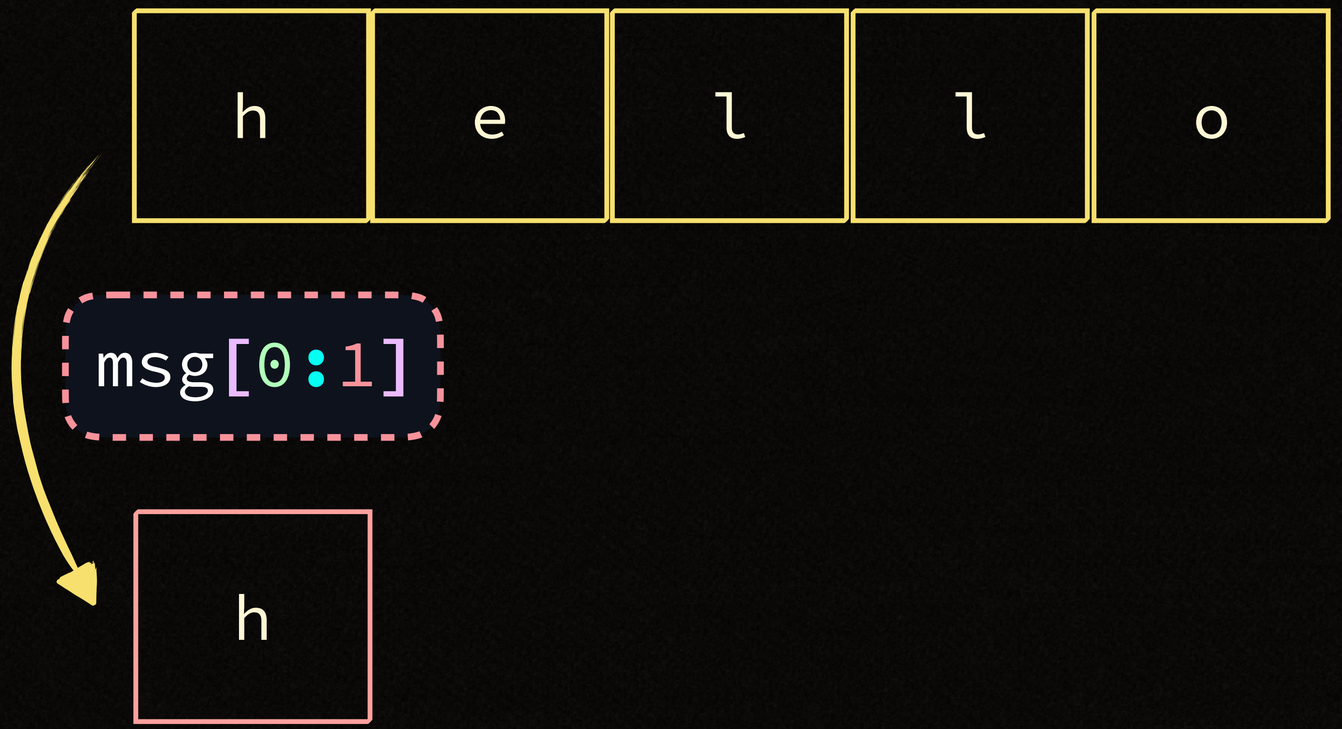


Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Creates a new slice from 0th index to the first element

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

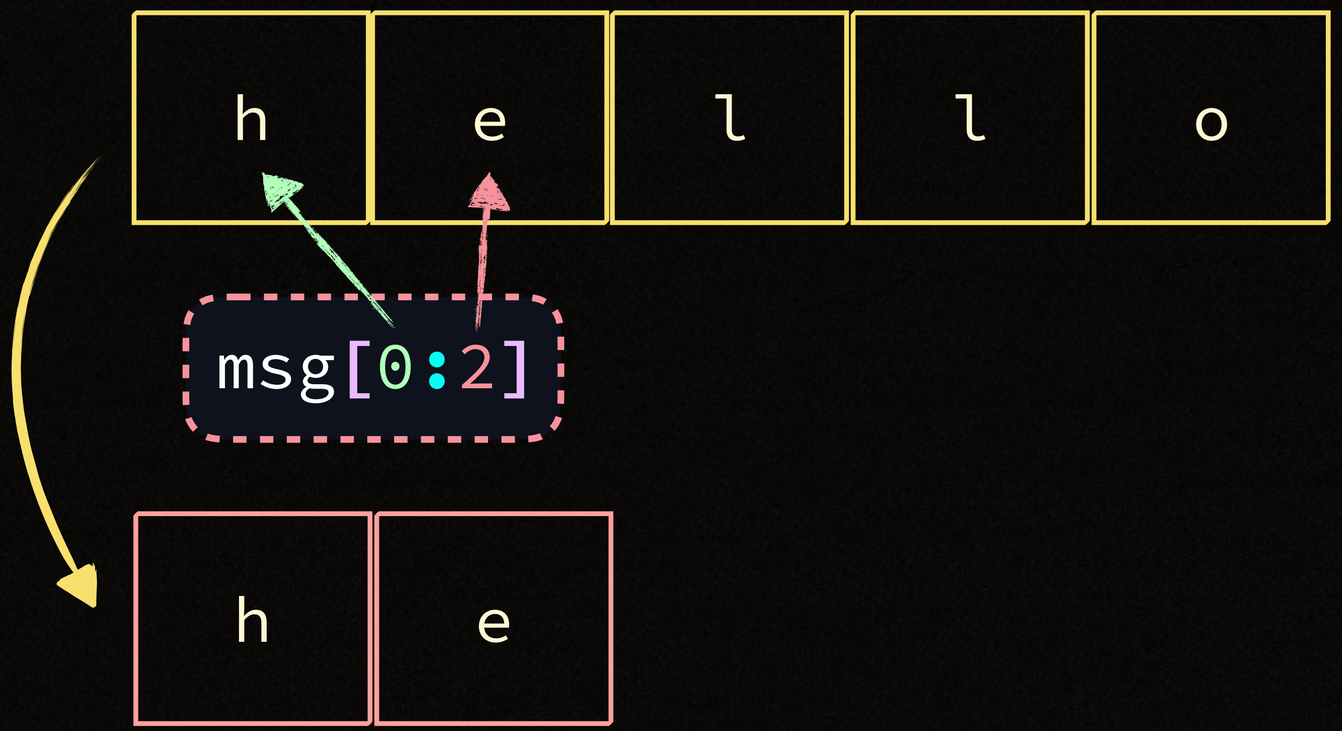


Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Creates a new slice from 0th index to the 2nd element

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

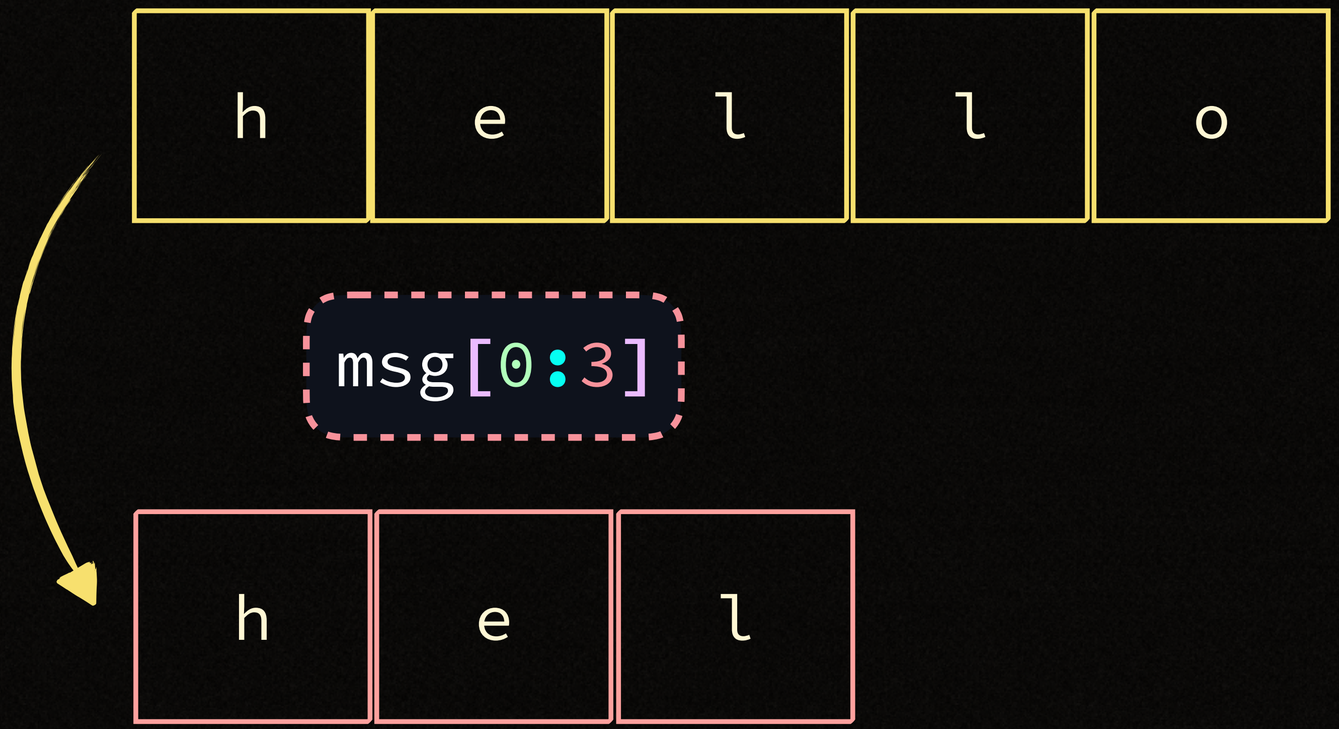


Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Creates a new slice from 0th index to the 3rd element

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Creates a new slice from 0th index to the 4th element

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Creates a new slice from 0th index to the last element

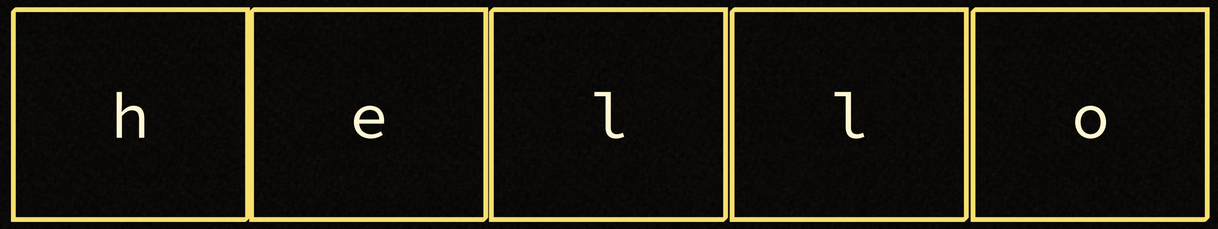
```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

You cannot slice a slice beyond its length *(not completely true)*

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```




```
msg[:6]
```

Runtime Error

Slice bounds out of range

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Creates a new slice from 1st index to the 4th element

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Creates a new slice from 1st index to the last element

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

Creates a new slice from 0th index to the 4th element

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appending to a slice appends after the last element

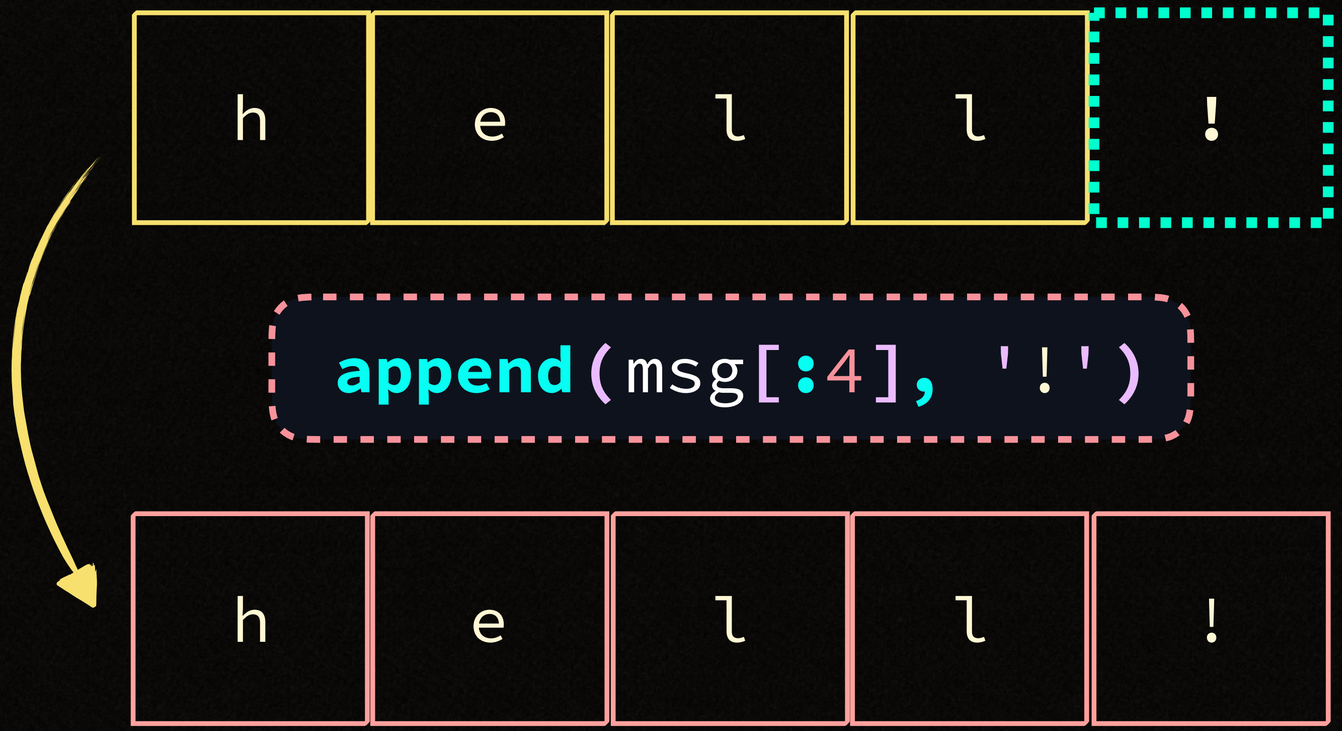
```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appending to a sliced slice overwrites the original slice

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE INTERNALS

BACKING ARRAY OF A SLICE

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

A slice doesn't *directly* store any elements

```
ages := []int{35, 15, 25}
```

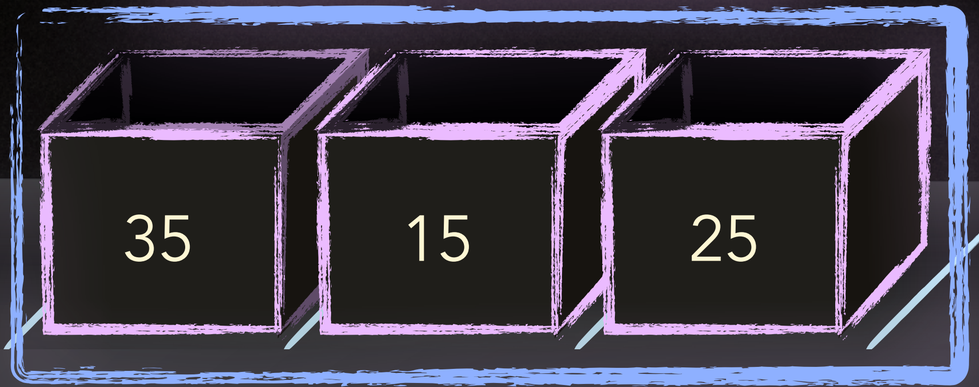
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

Slice literal creates a hidden array and returns a slice that refers to that array

```
ages := []int{35, 15, 25}
```

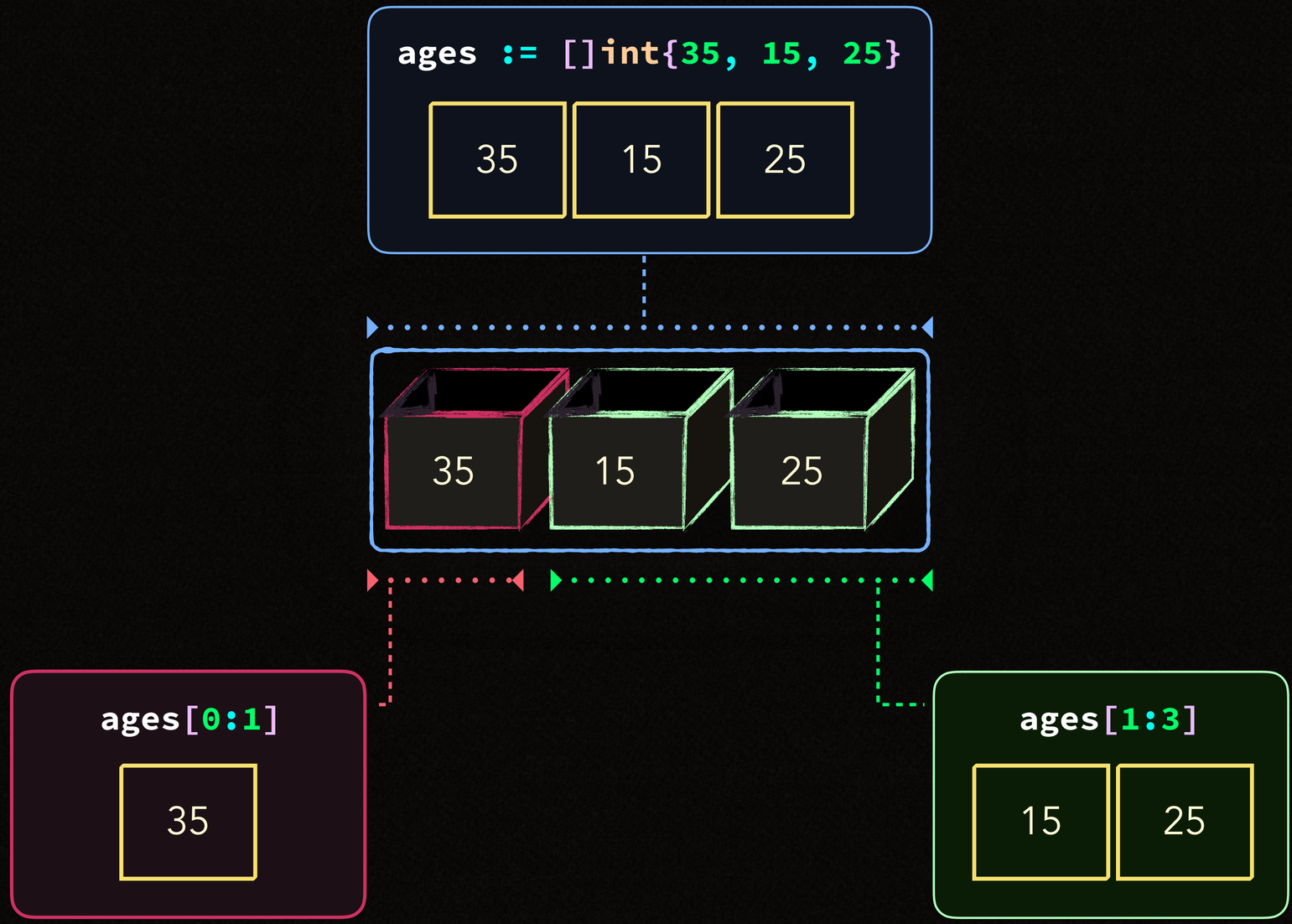
BACKING ARRAY



the real storage stored separately from the slice

BACKING ARRAY

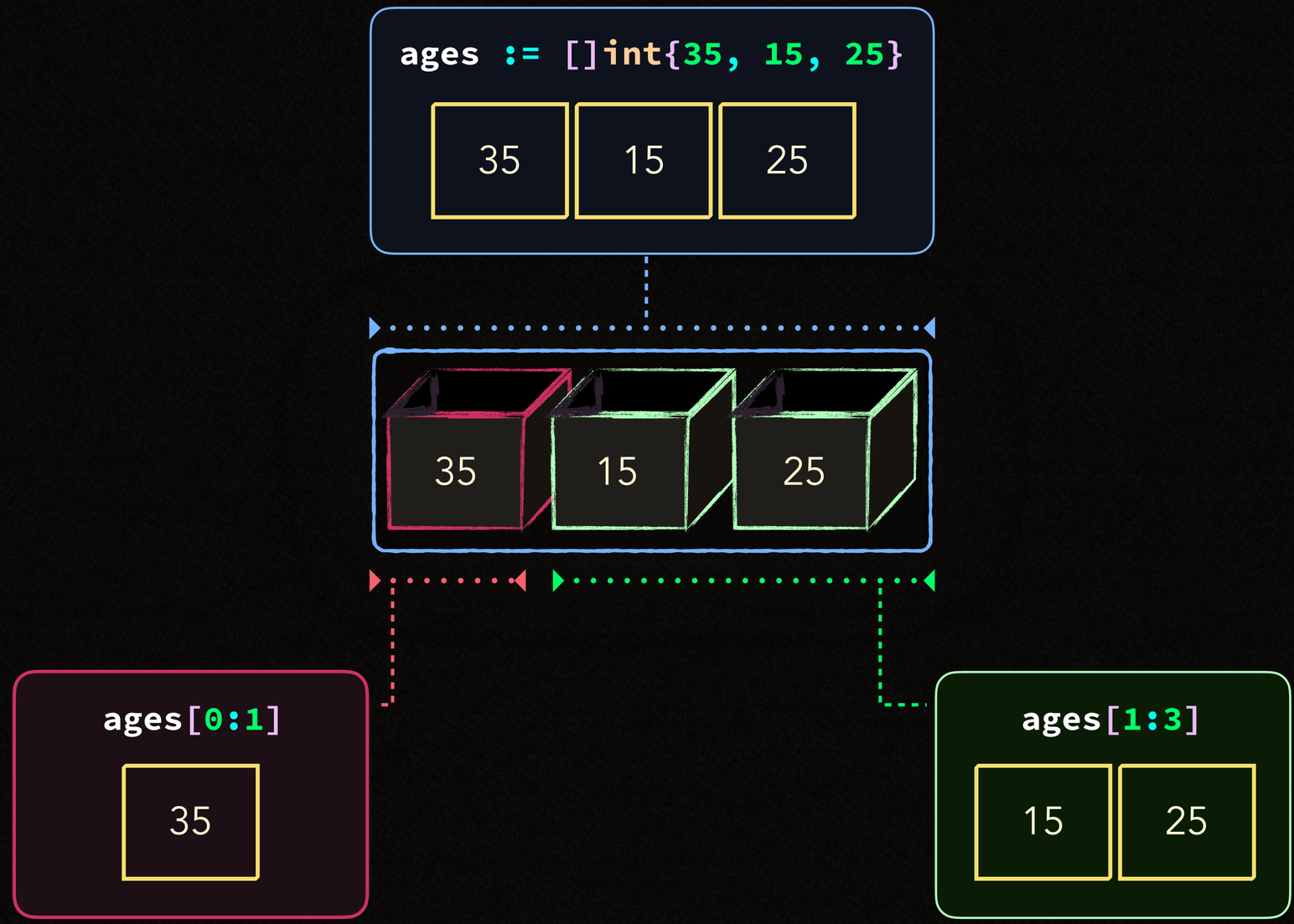
A slice is a window to its backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

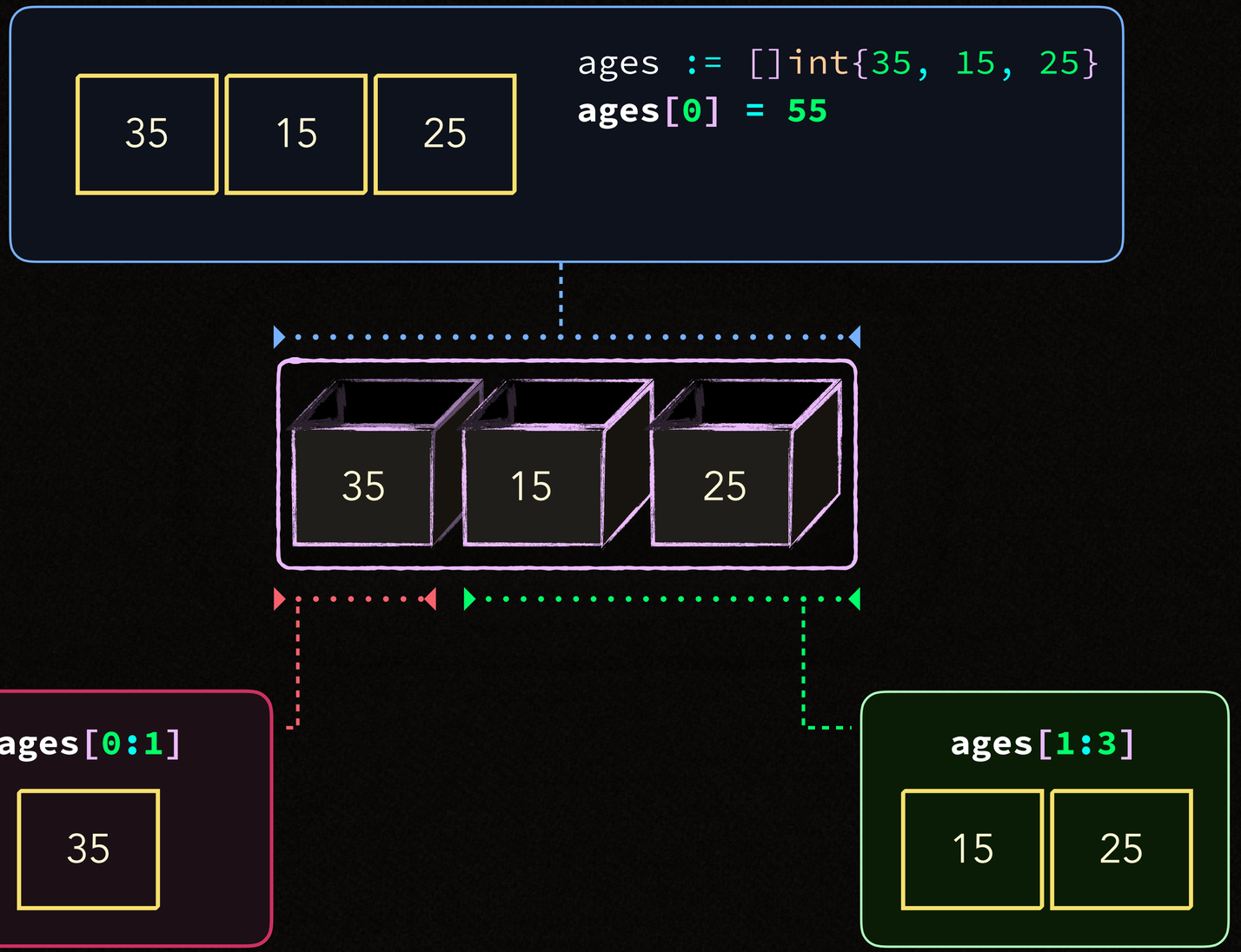
A slice created with a slice expression share the same array with the original slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

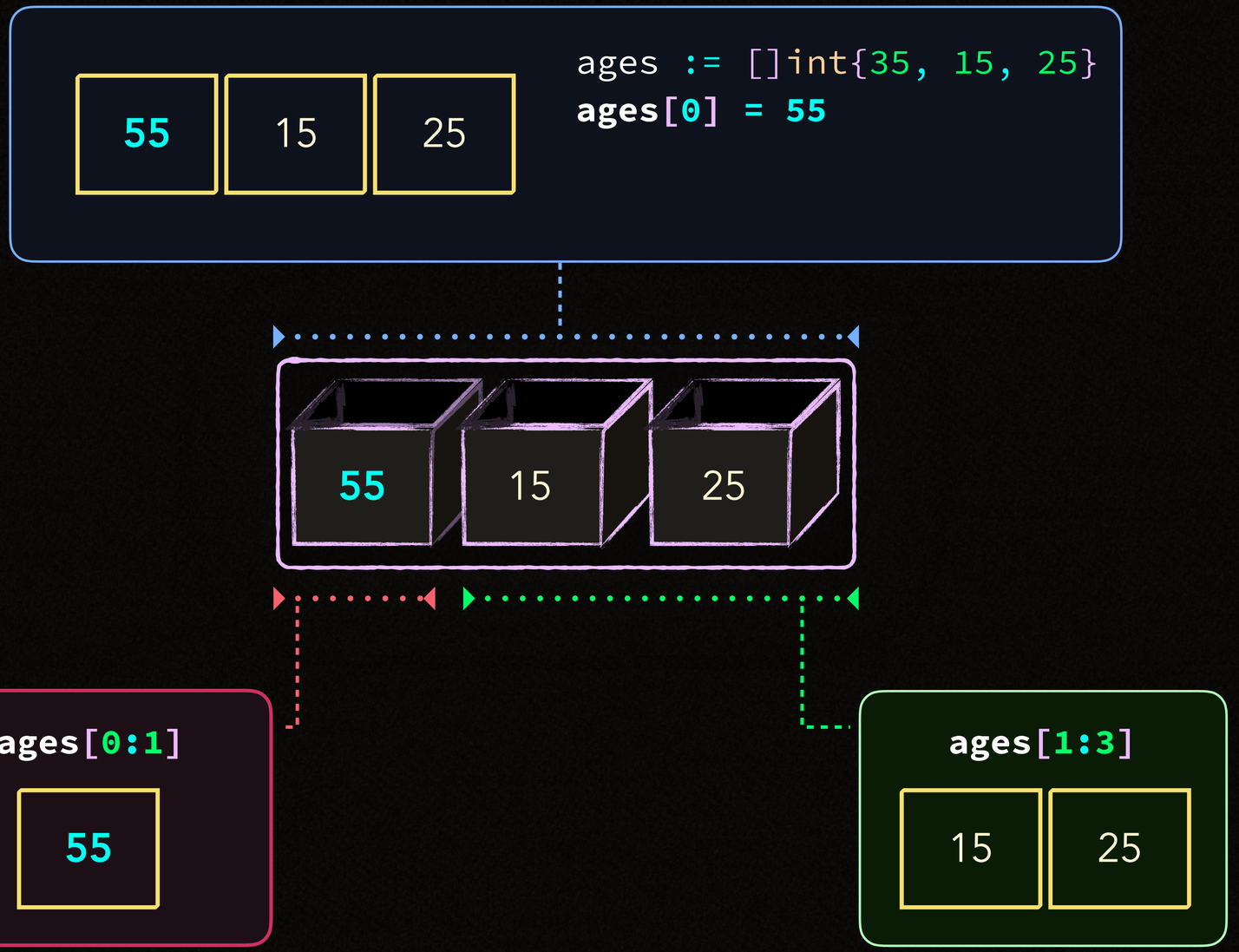
Slices see the changes in the shared backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

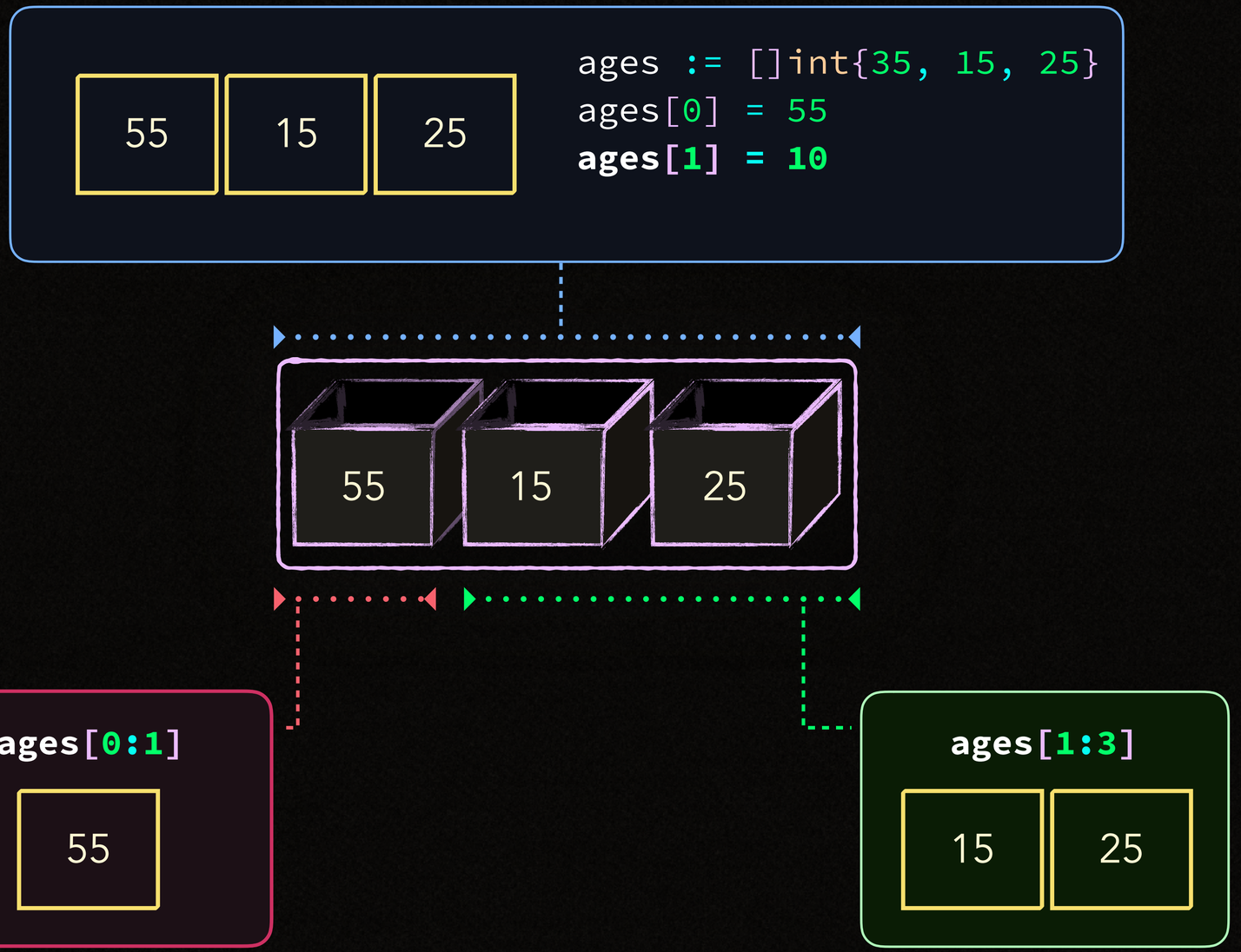
Slices see the changes in the shared backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

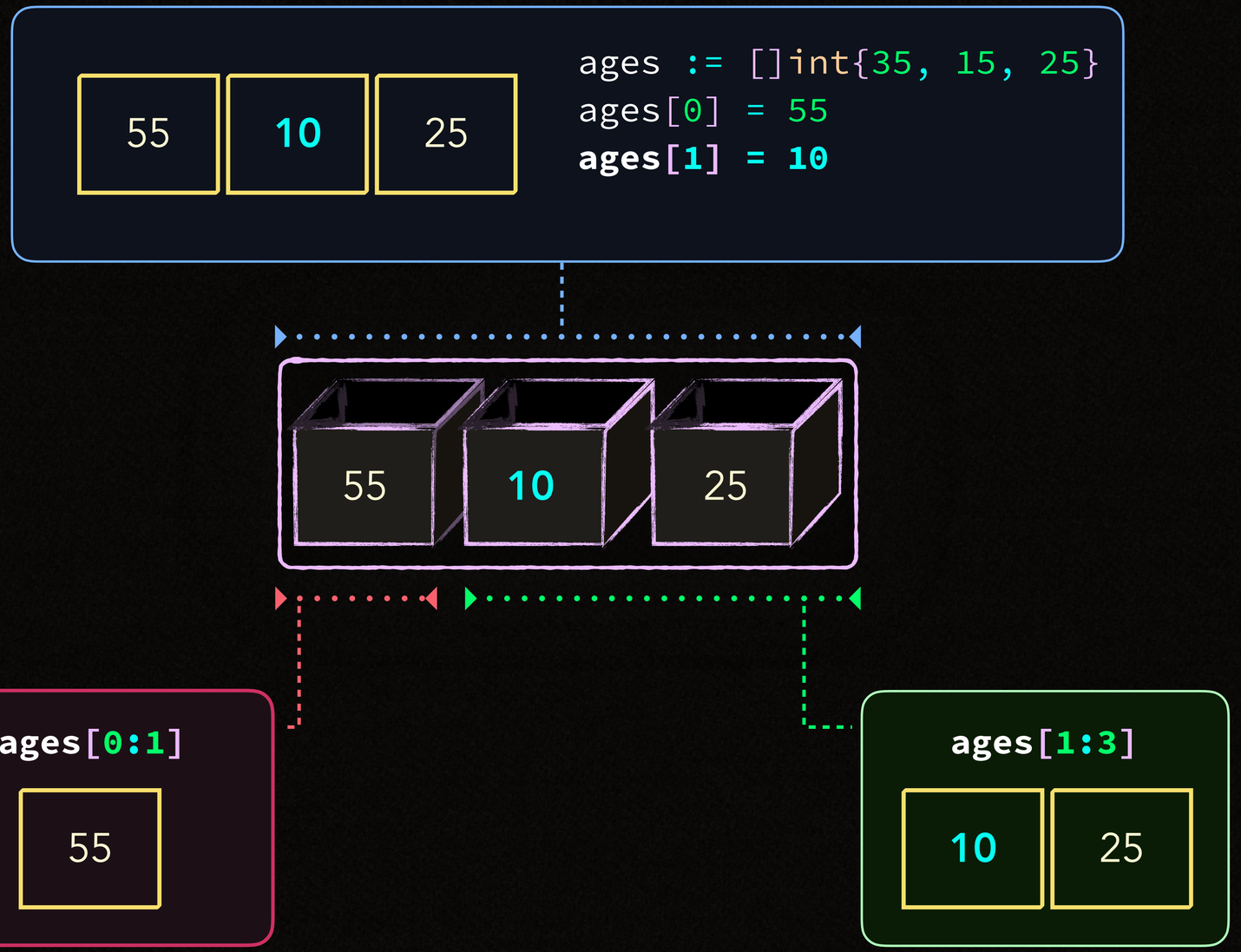
Slices see the changes in the shared backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

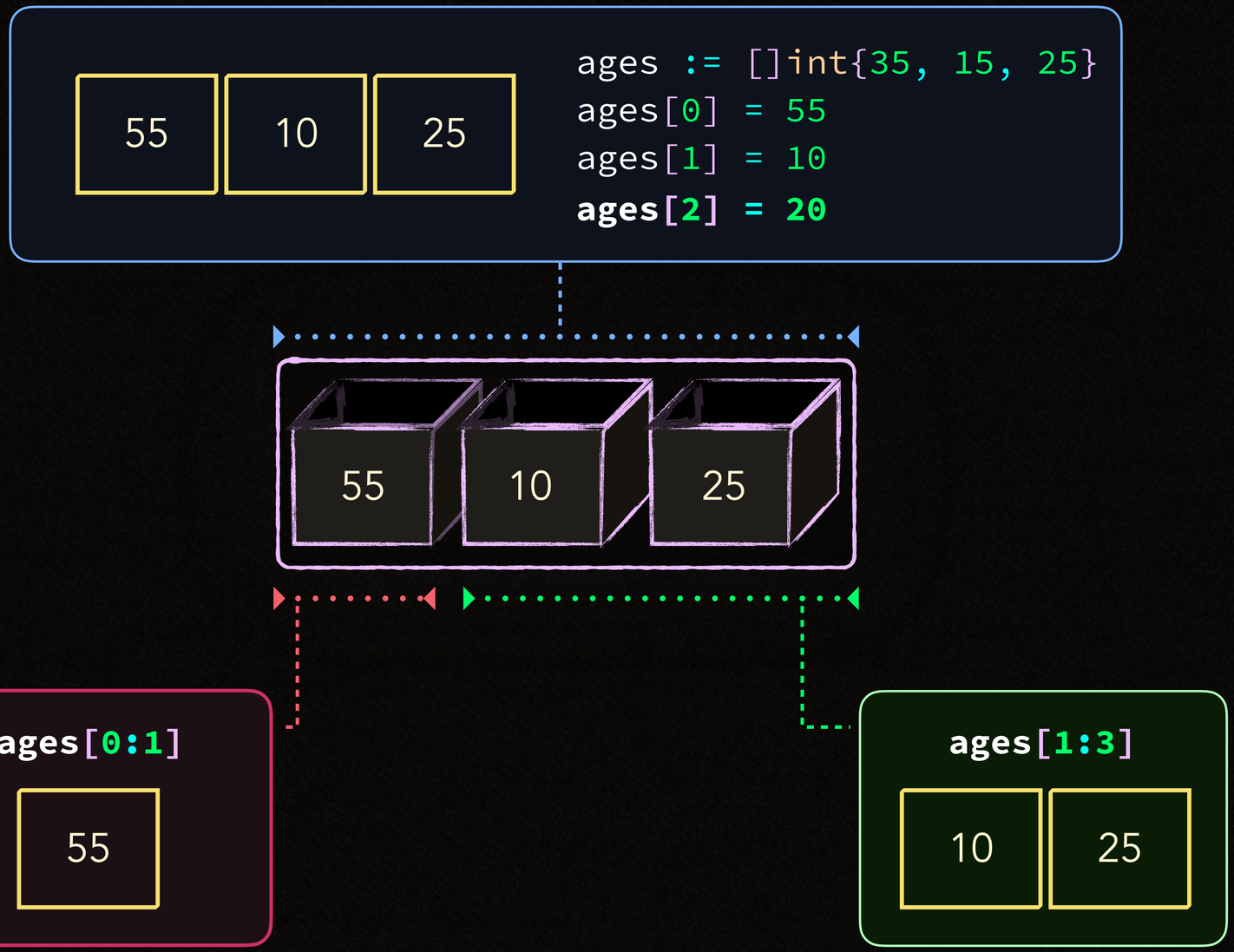
Slices see the changes in the shared backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

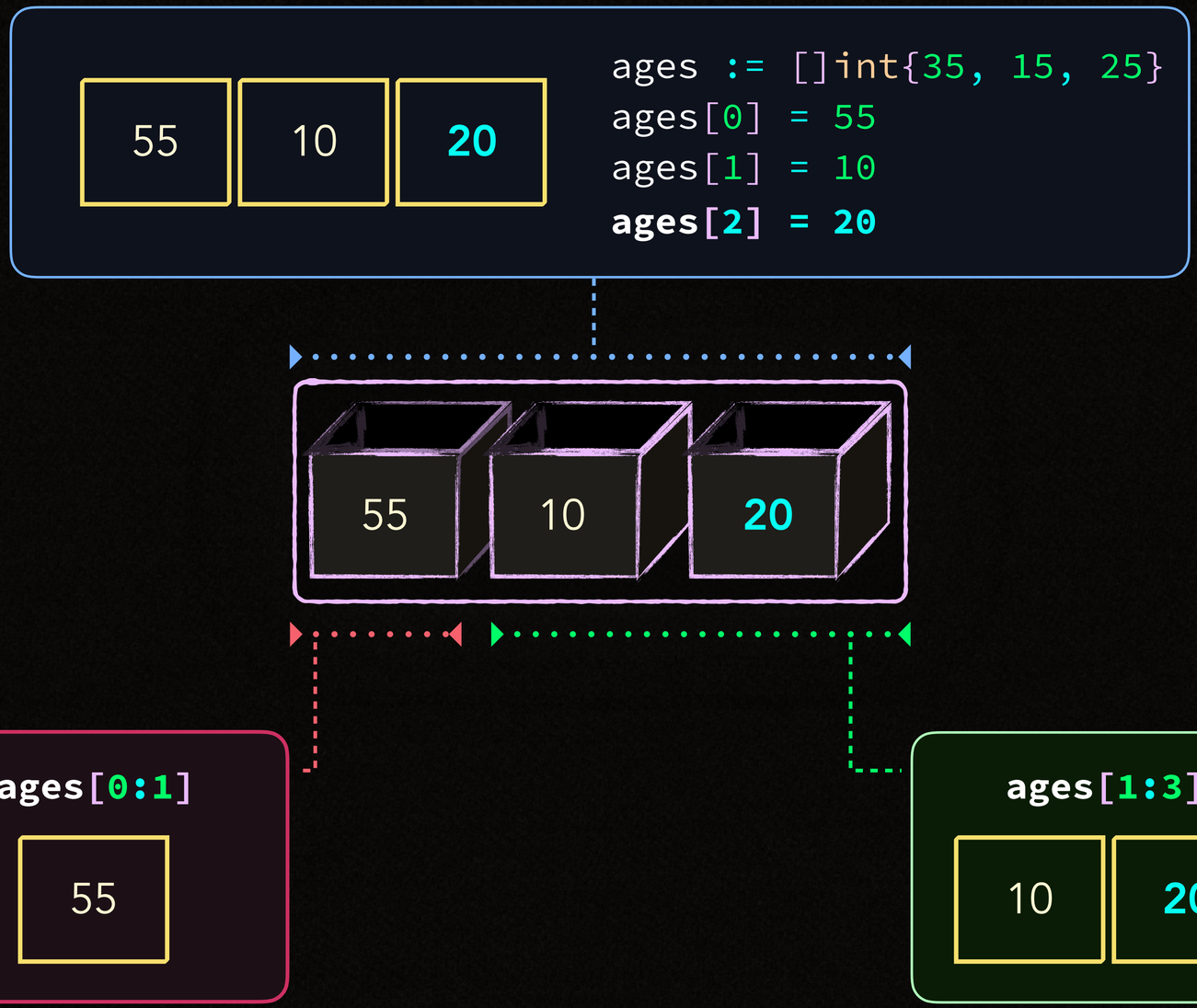
Slices see the changes in the shared backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

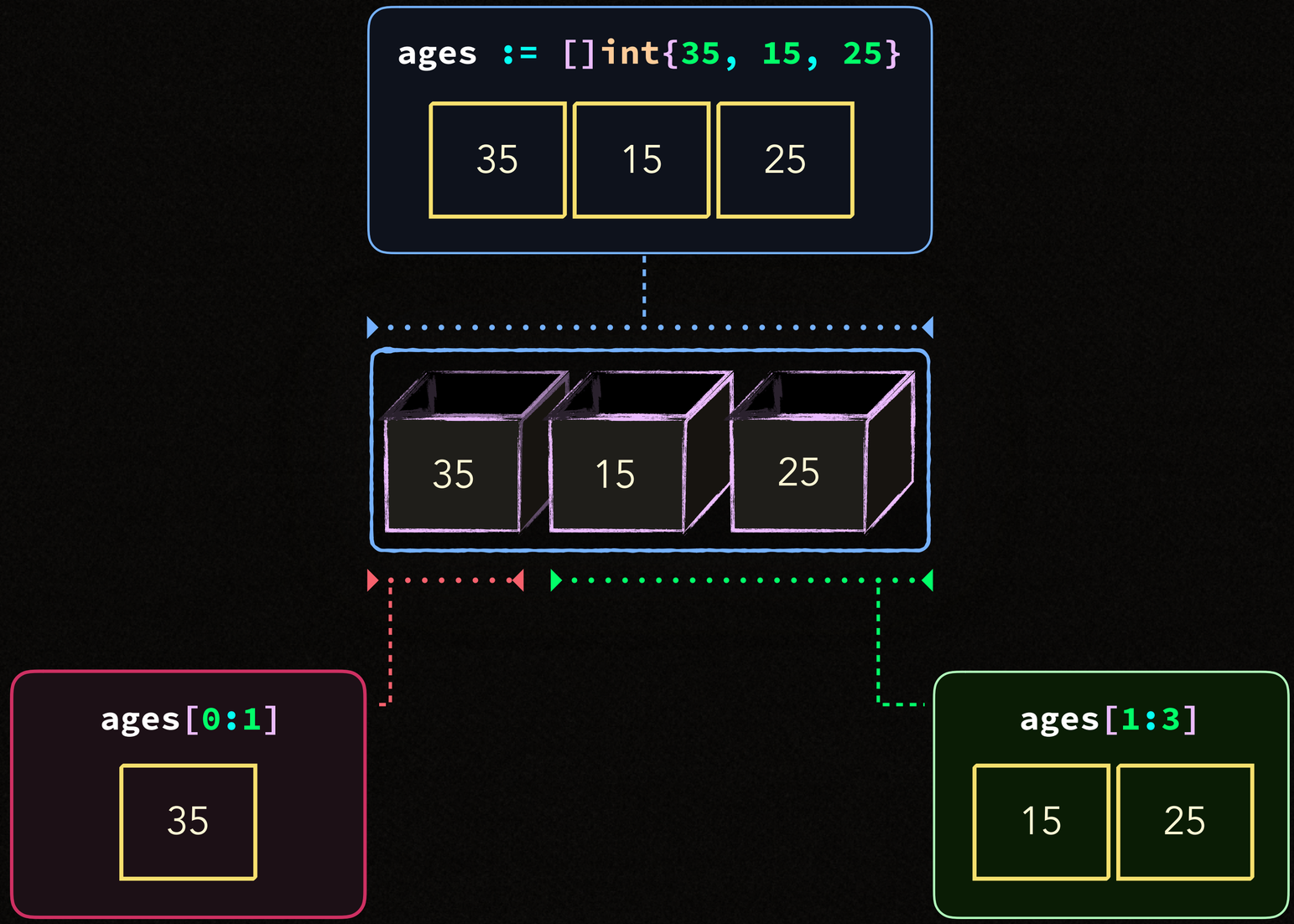
Slices see the changes in the shared backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

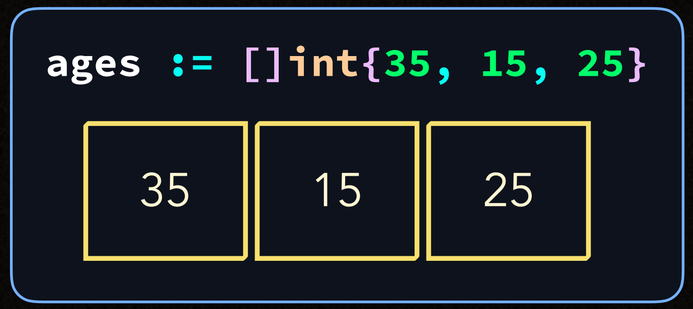
Slicing is cheap because of the same backing array is used

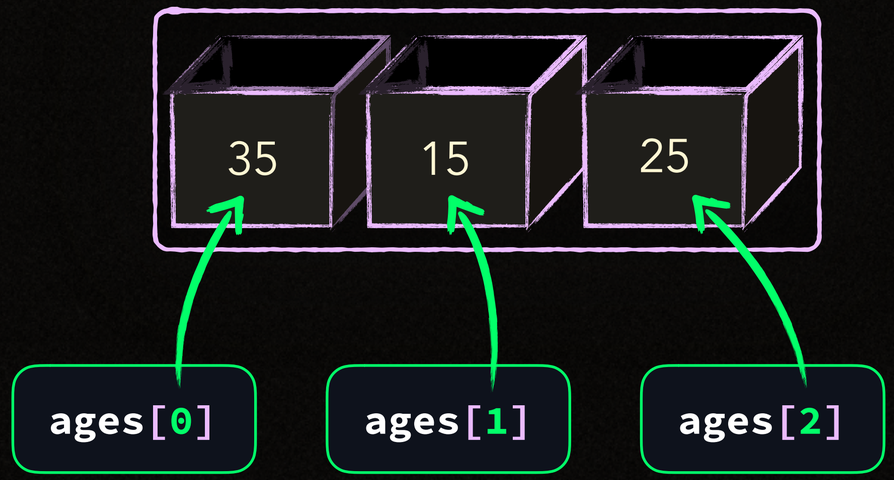


Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

Indexing a slice is fast because the slice's backing array is contiguous in memory



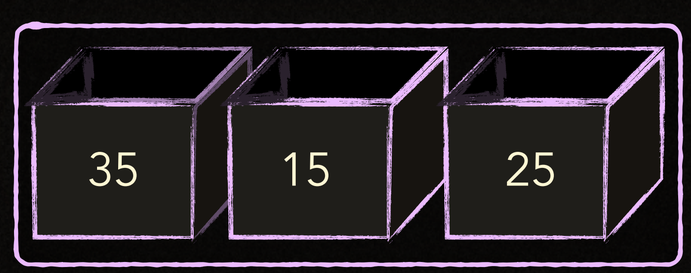


Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

Slices can have separate backing arrays

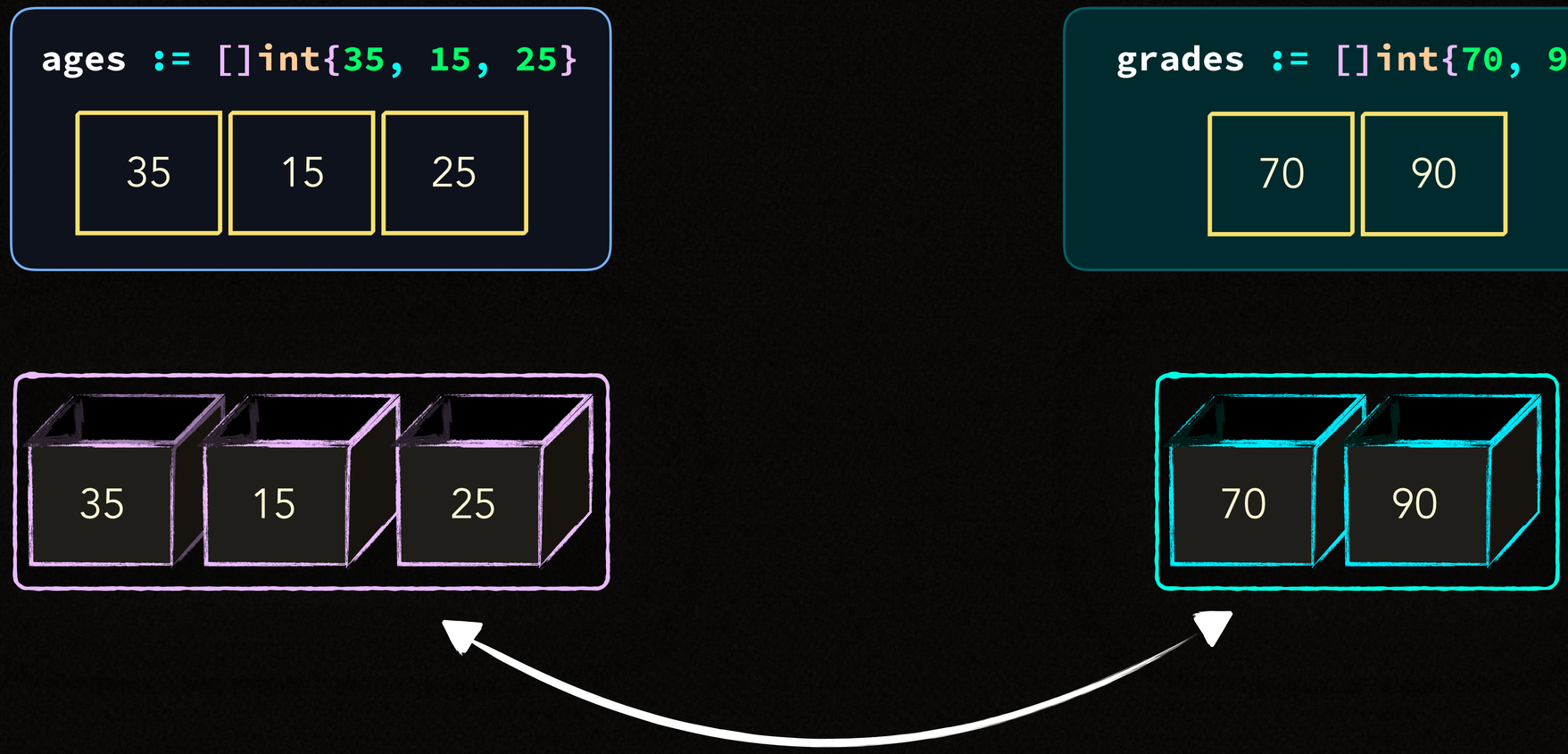
```
ages := []int{35, 15, 25}
```



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

A non-empty slice literal always creates a new backing array

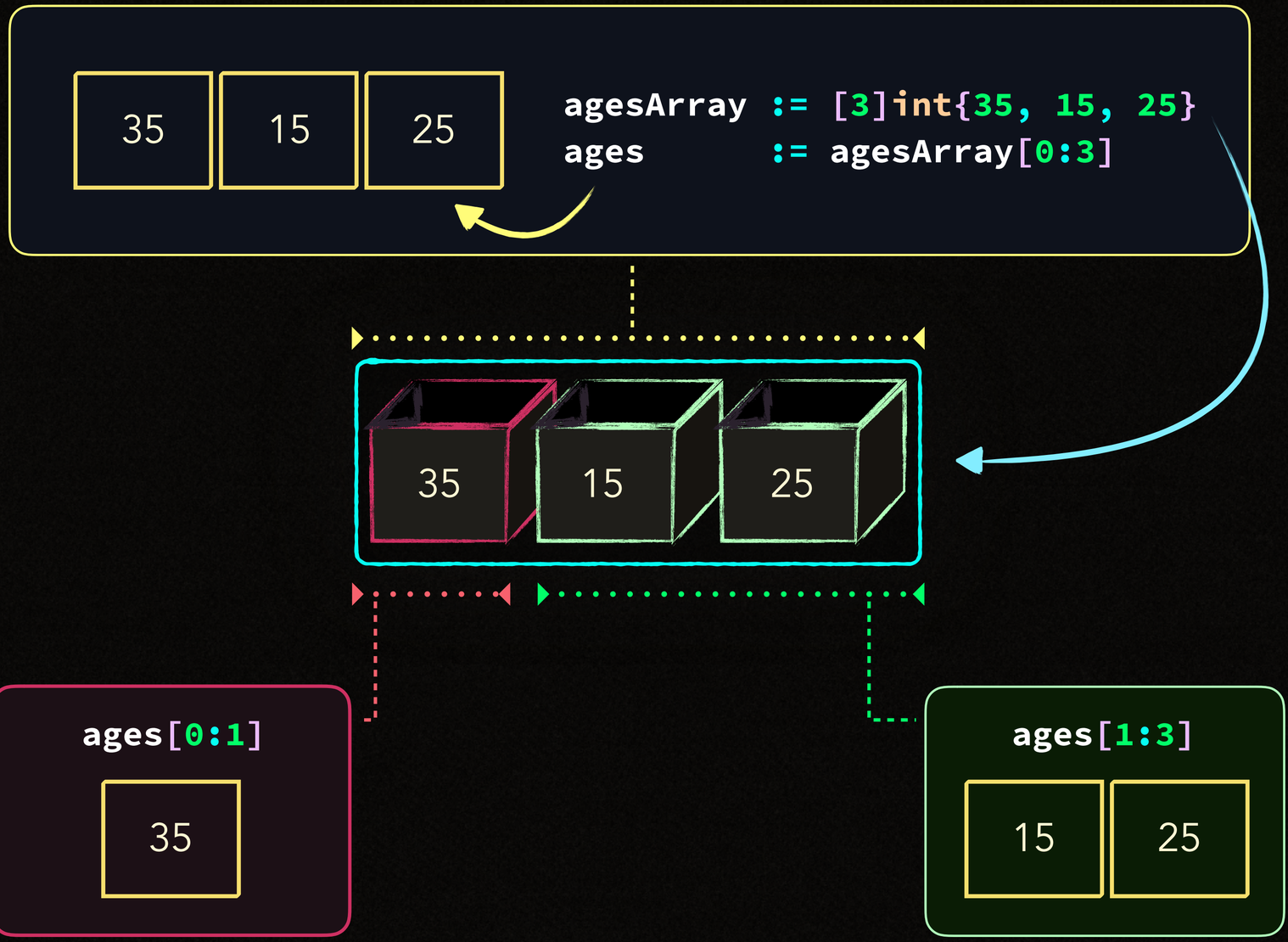


These slices have separate backing arrays

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

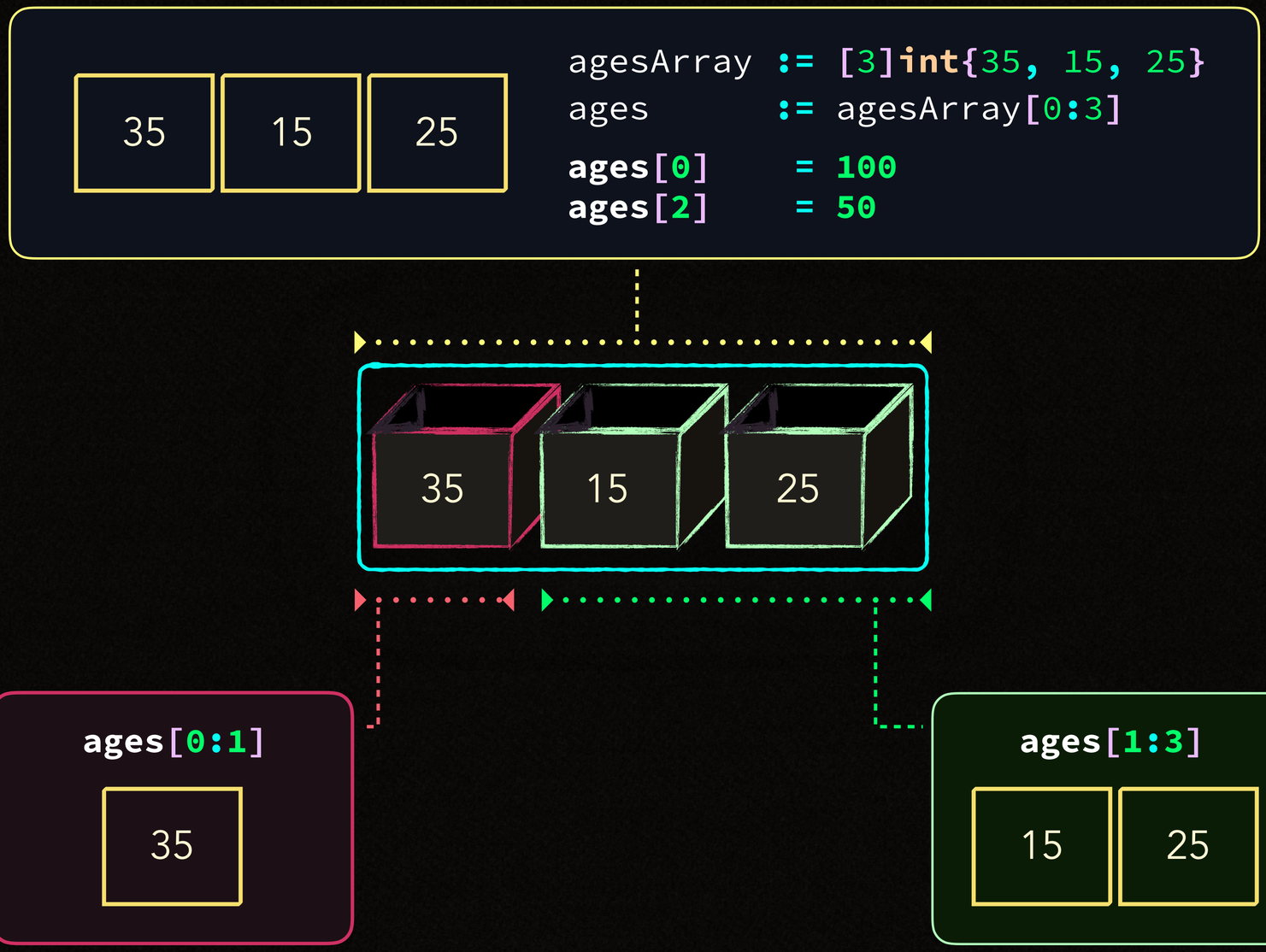
A "sliced array" becomes the backing array of the returned slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

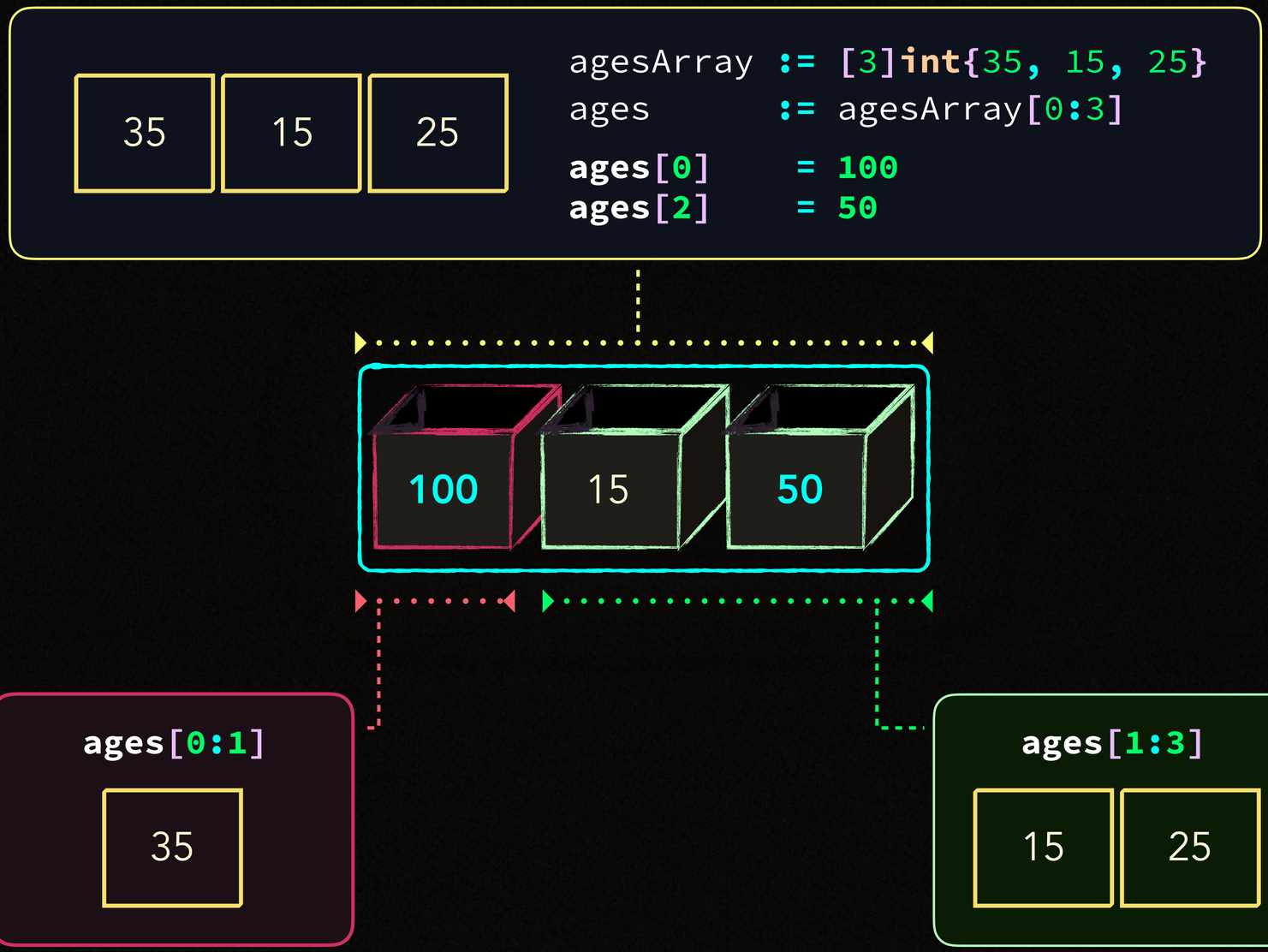
A "sliced array" becomes the backing array of the returned slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

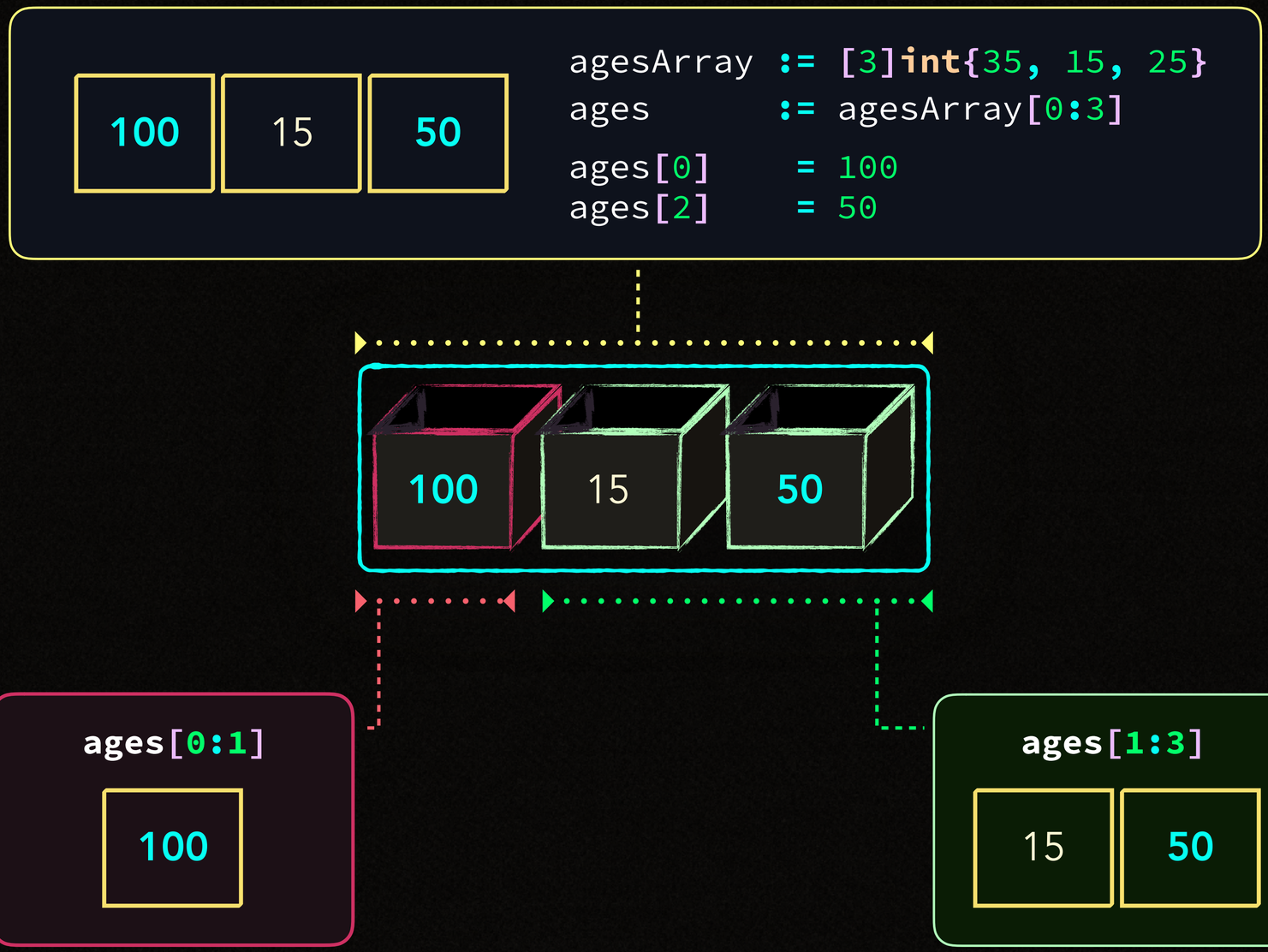
A sliced array is shared among the slices as their backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

A sliced array is shared among the slices as their backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

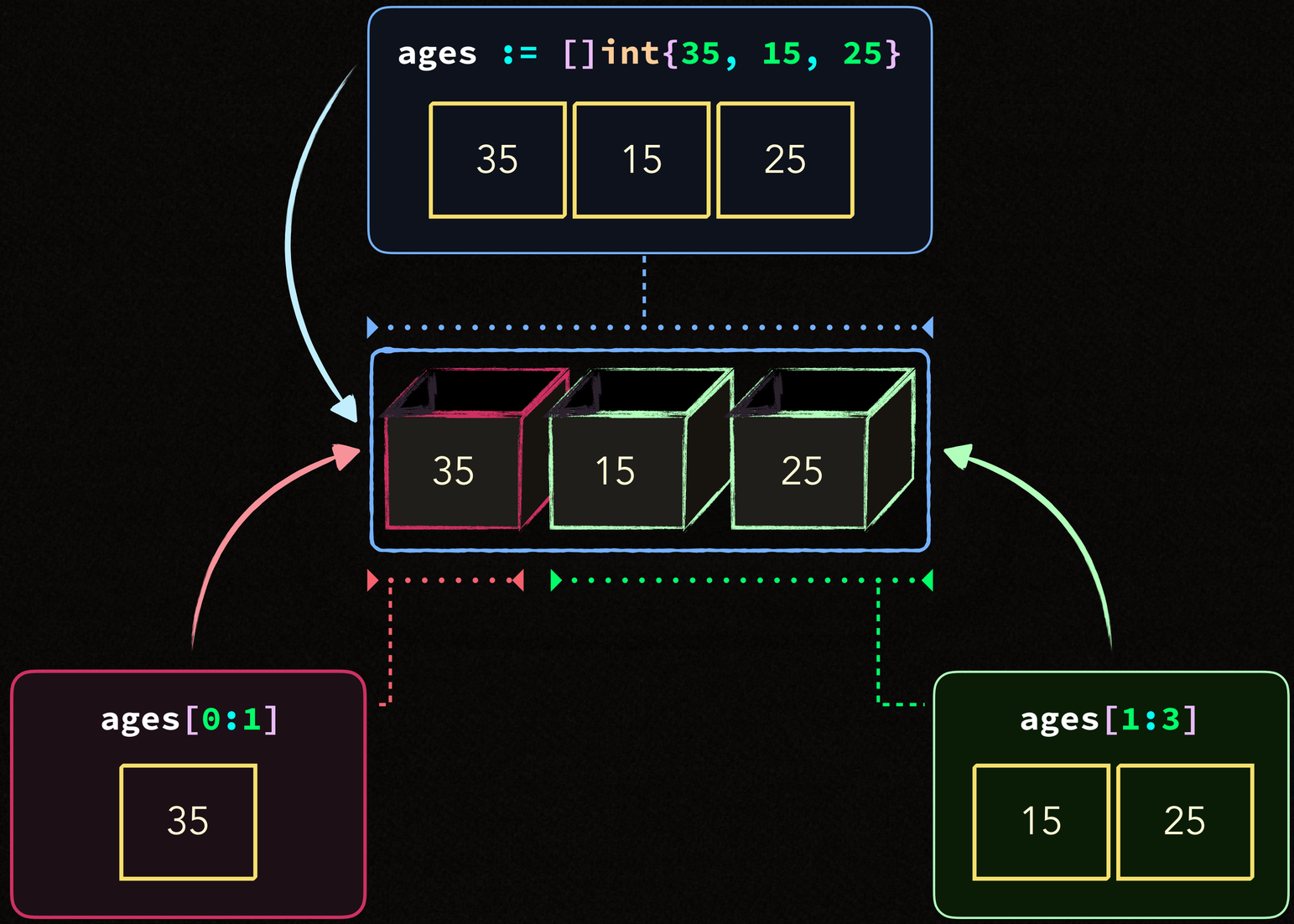
SLICE INTERNALS

SLICE HEADER

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE VALUE

If a slice value doesn't store any elements, so what does it contain?



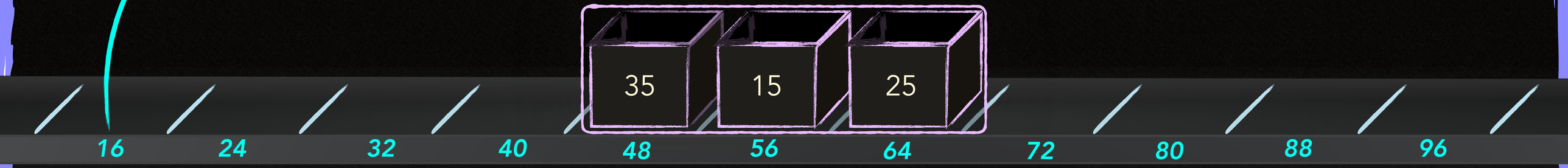
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE VALUE

A slice value refers to its backing array through a memory address

```
ages := []int{35, 15, 25}
```

Computer memory addresses



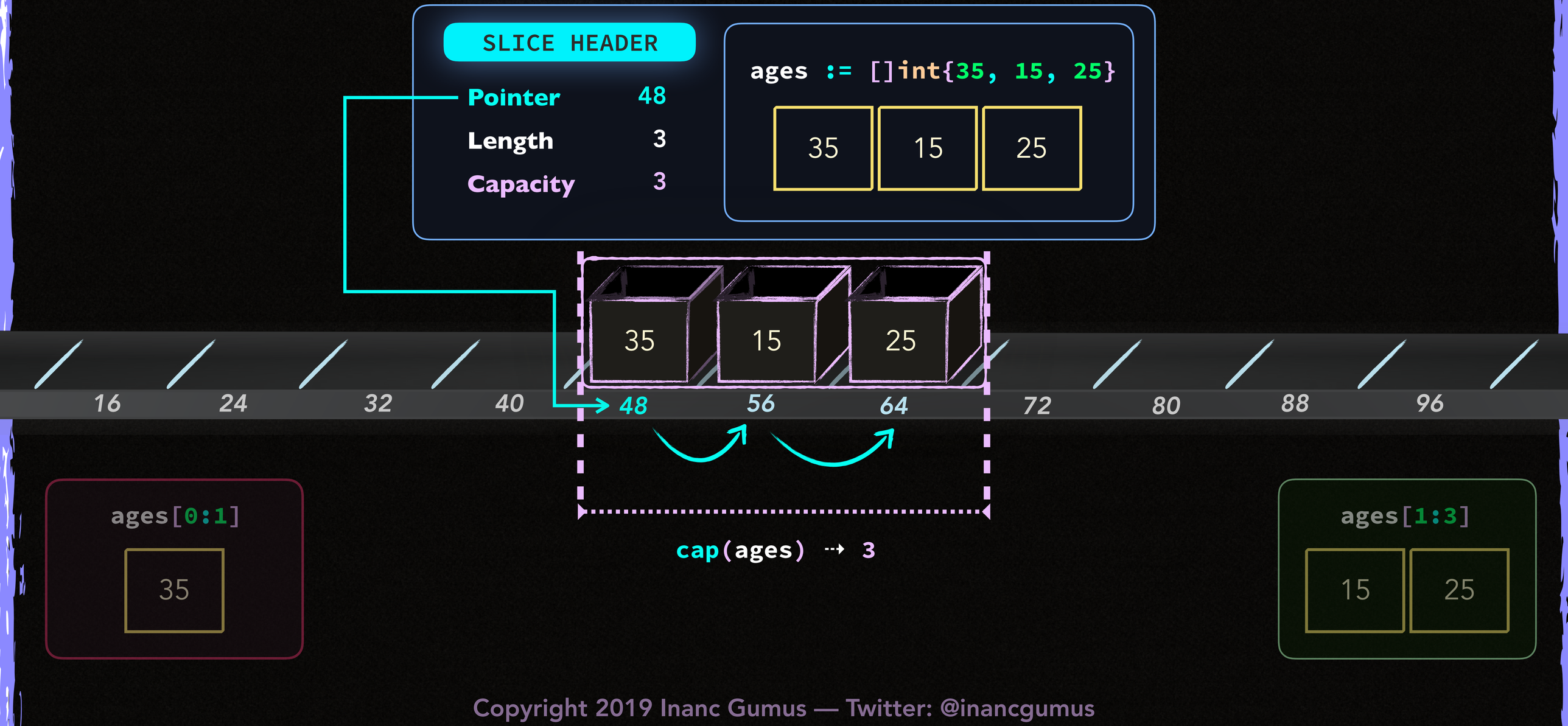

```
ages[0:1]
```

```
ages[1:3]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE HEADER

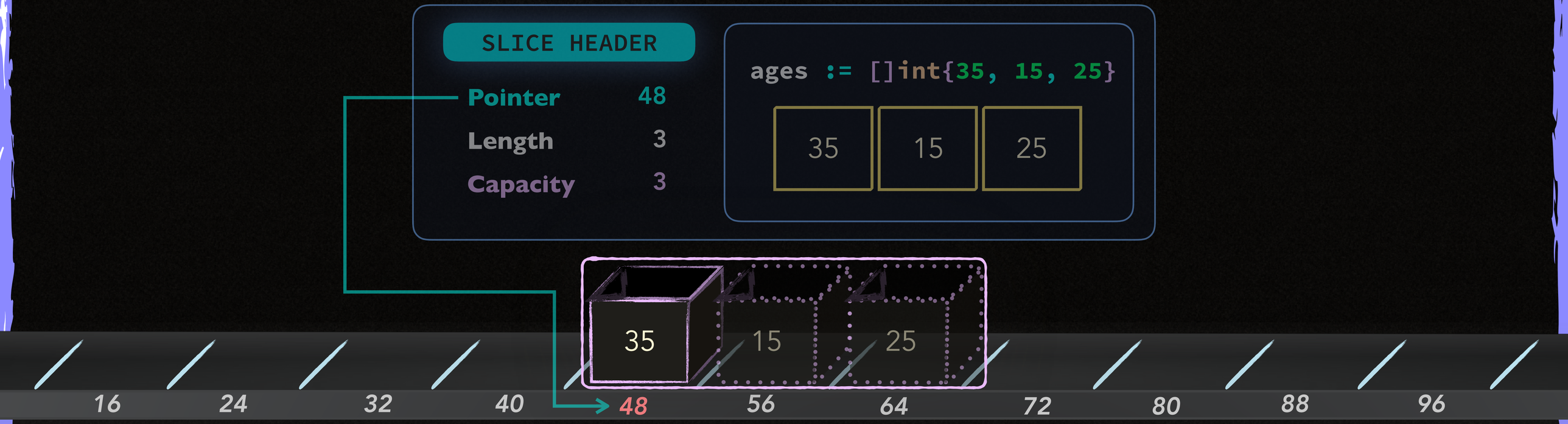
Slice = SliceHeader, it's a tiny data structure that describes its backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE HEADER

A slice cannot index elements beyond its length *(without reslicing)*

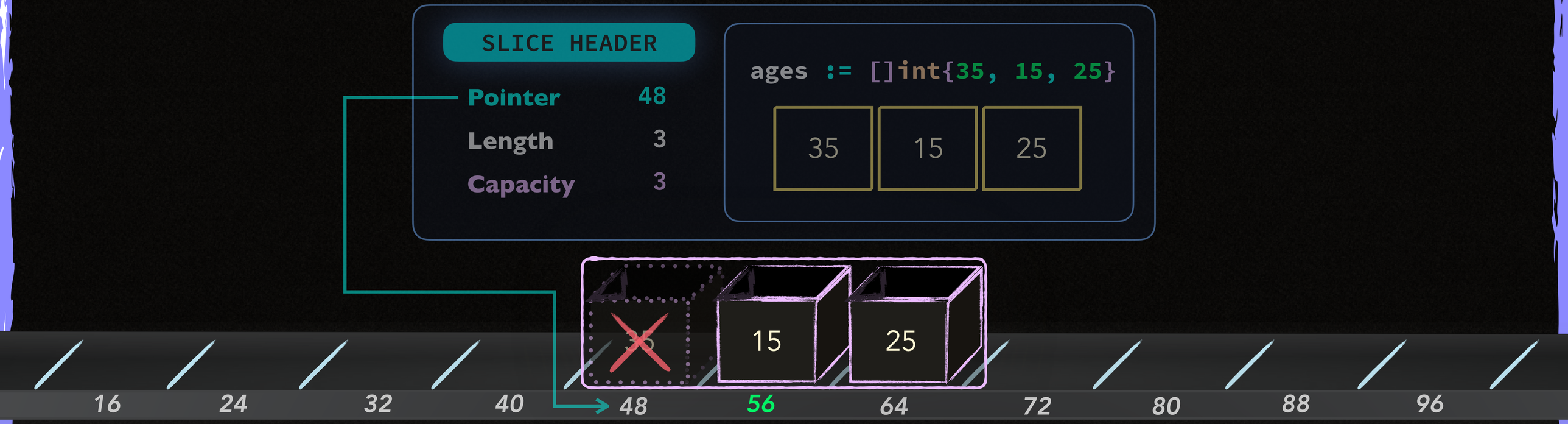


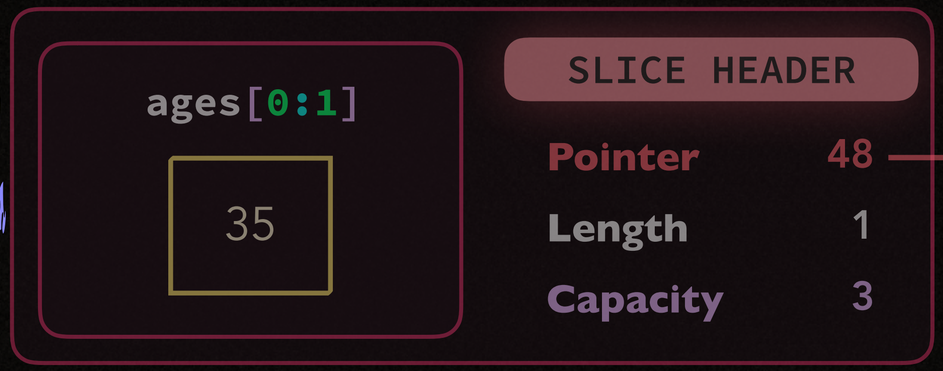
```
red := ages[0:1]
red[0] ⇢ 35
red[1] ⇢ X
red[2] ⇢ X
```

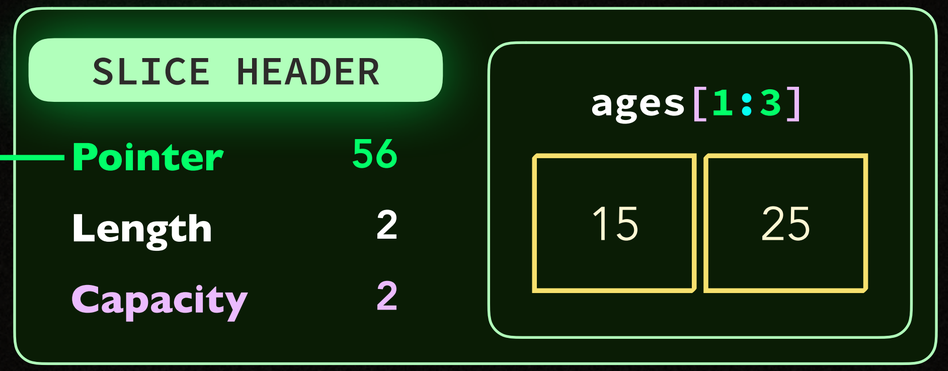
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE HEADER

Elements that are located before the slice's pointer remain invisible to the slice



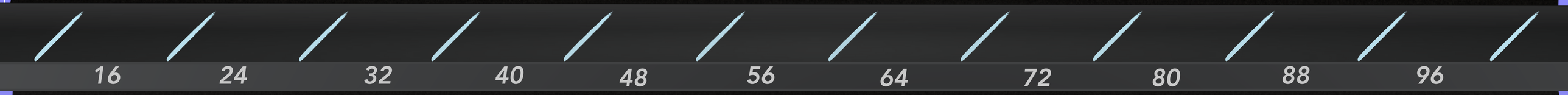




Copyright 2019 Inanc Gumus — Twitter: @inancgumus

NIL SLICE

An nil slice doesn't have a backing array but it has a slice header



When you don't know whether a **slice** will ever have any elements

declare it as a nil slice

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE INTERNALS

CAPACITY

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

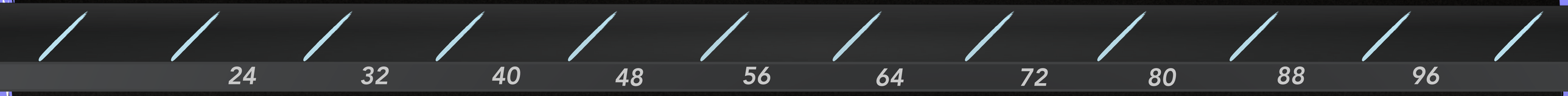
NIL SLICE

A nil slice don't have a backing array so its capacity is zero

SLICE HEADER	
Pointer	0
Length	0
Capacity	0

```
len(ages) ⇢ 0

cap(ages) ⇢ 0
```




```
var ages []int
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE LITERAL

This slice literal has 3 elements; so it creates an array with 3 elements as well

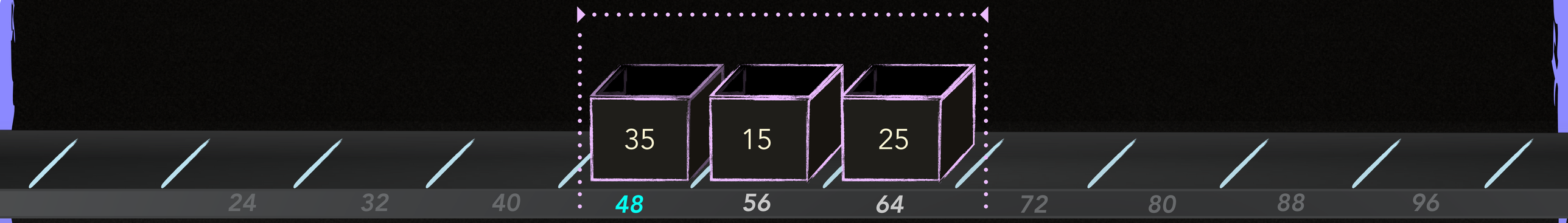
```
ages := []int{35, 15, 25}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE LITERAL

Capacity is the length of the backing array *(not the slice's)*

The length of this backing array is 3

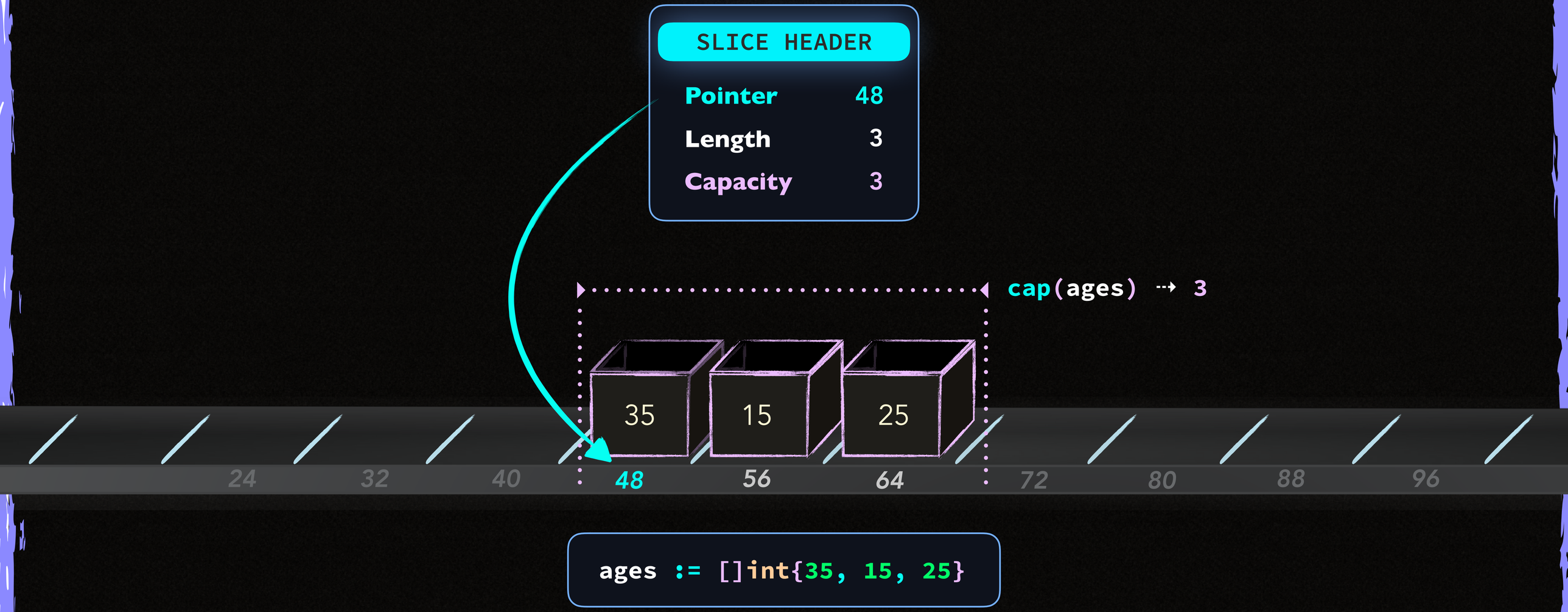



```
ages := []int{35, 15, 25}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

Capacity is all about the length of a backing array and where a slice starts

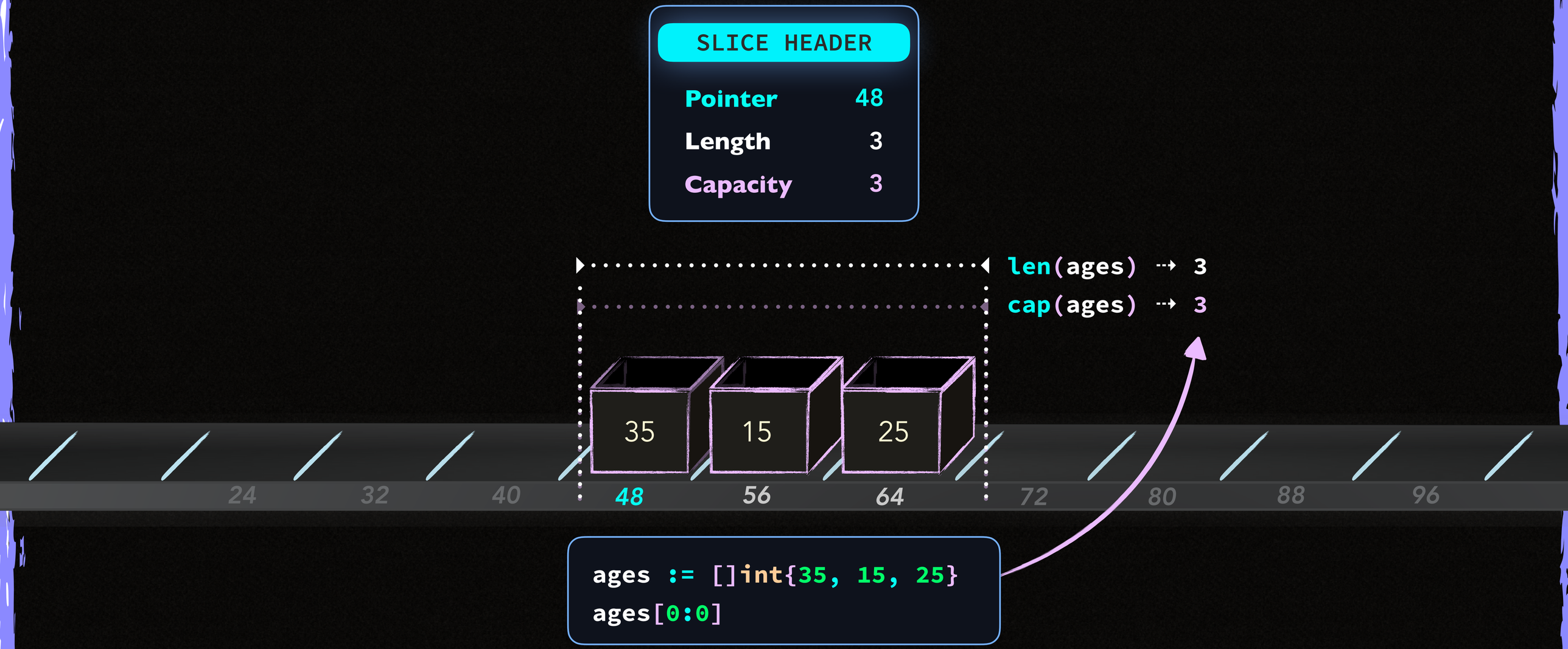



```
ages := []int{35, 15, 25}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

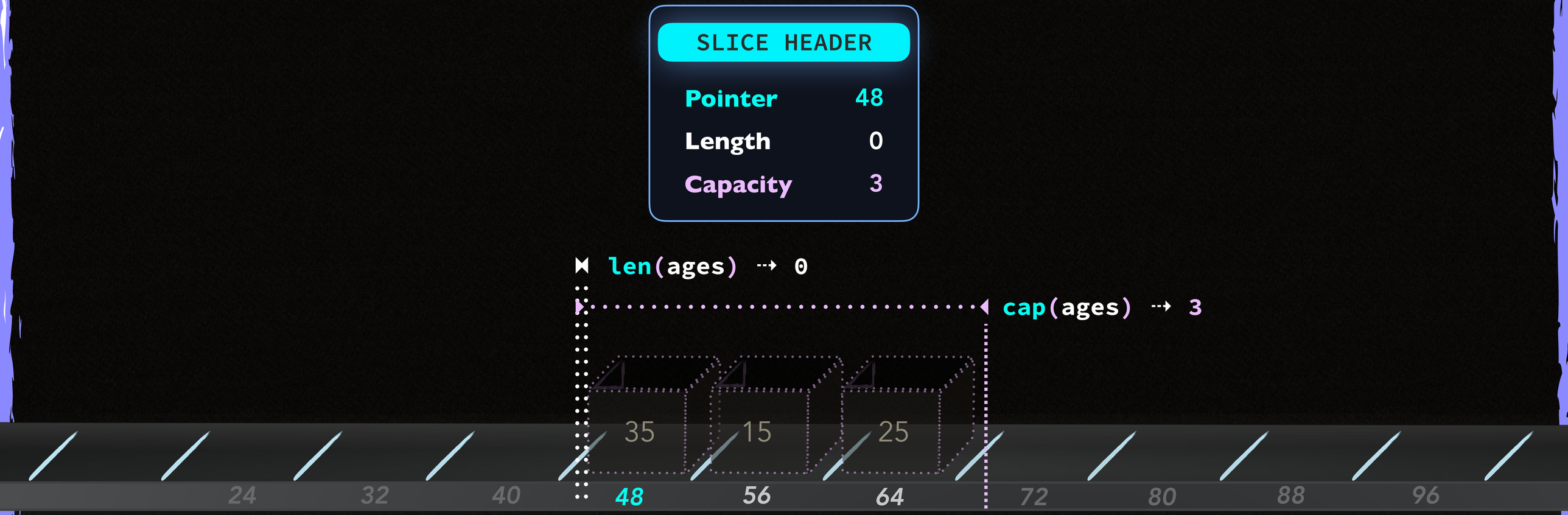
A slice created by a slice literal will have the same len and cap



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

This empty slice's length is zero but its backing array has 3 elements

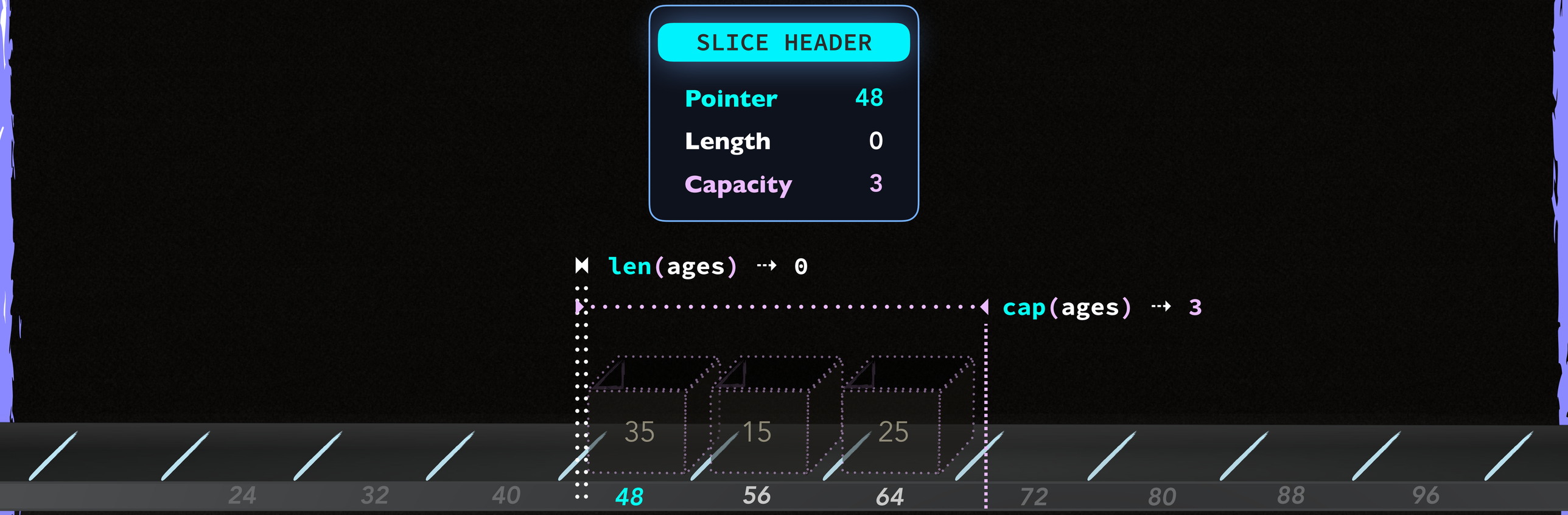



```
ages := []int{35, 15, 25}
ages[0:0]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

Capacity determines how much you can extend a slice

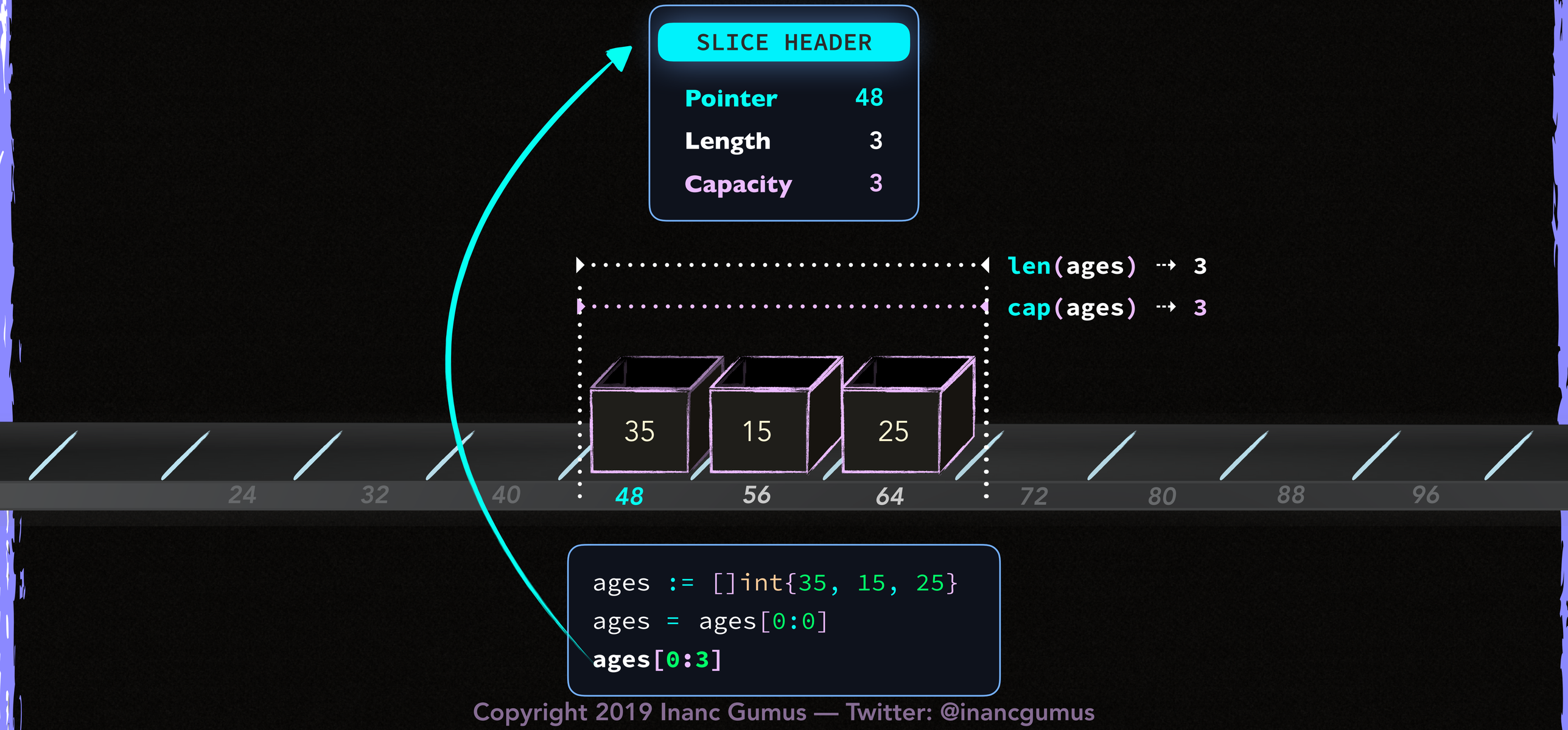



```
ages := []int{35, 15, 25}
ages = ages[0:0]
ages[0:3]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

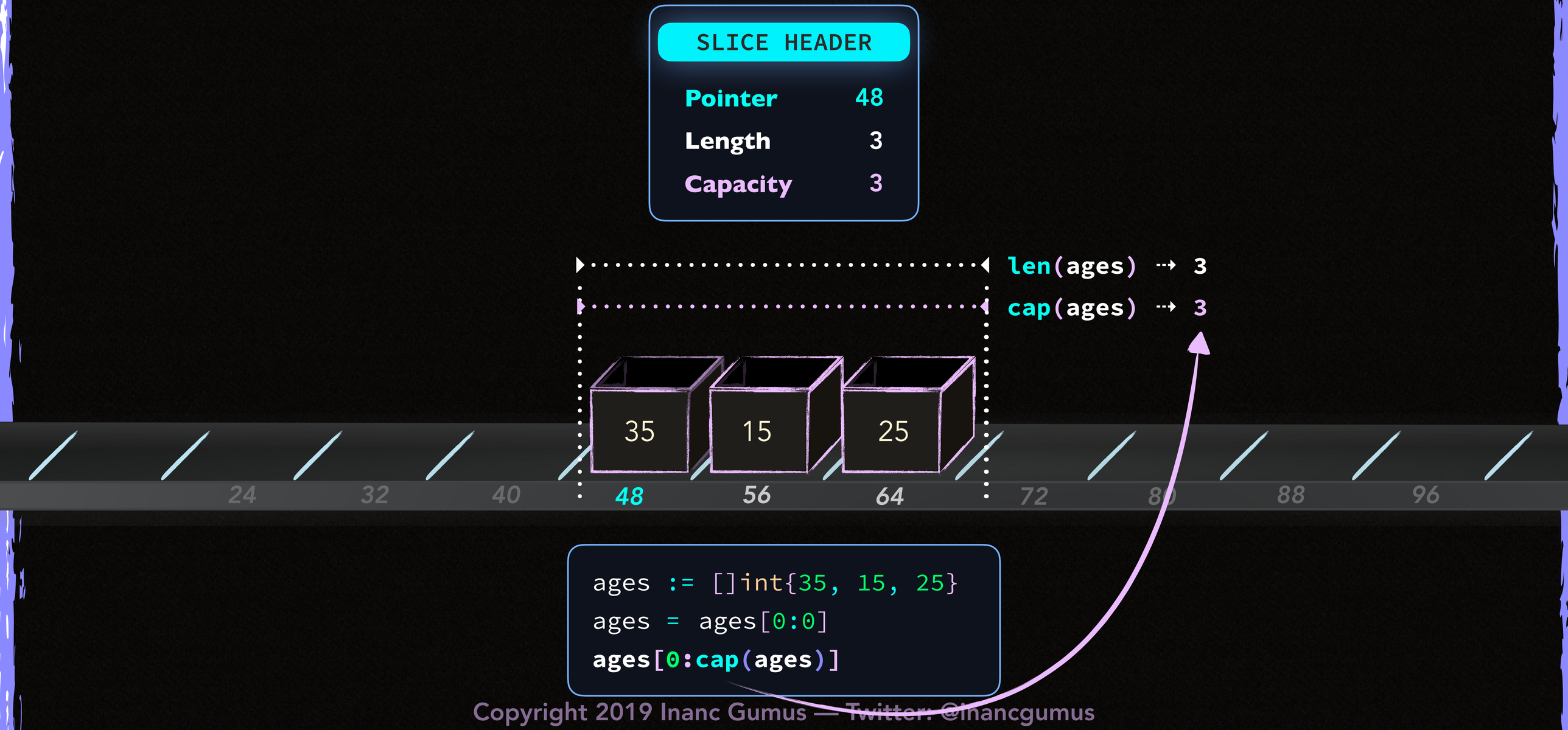
Capacity determines how much you can extend a slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

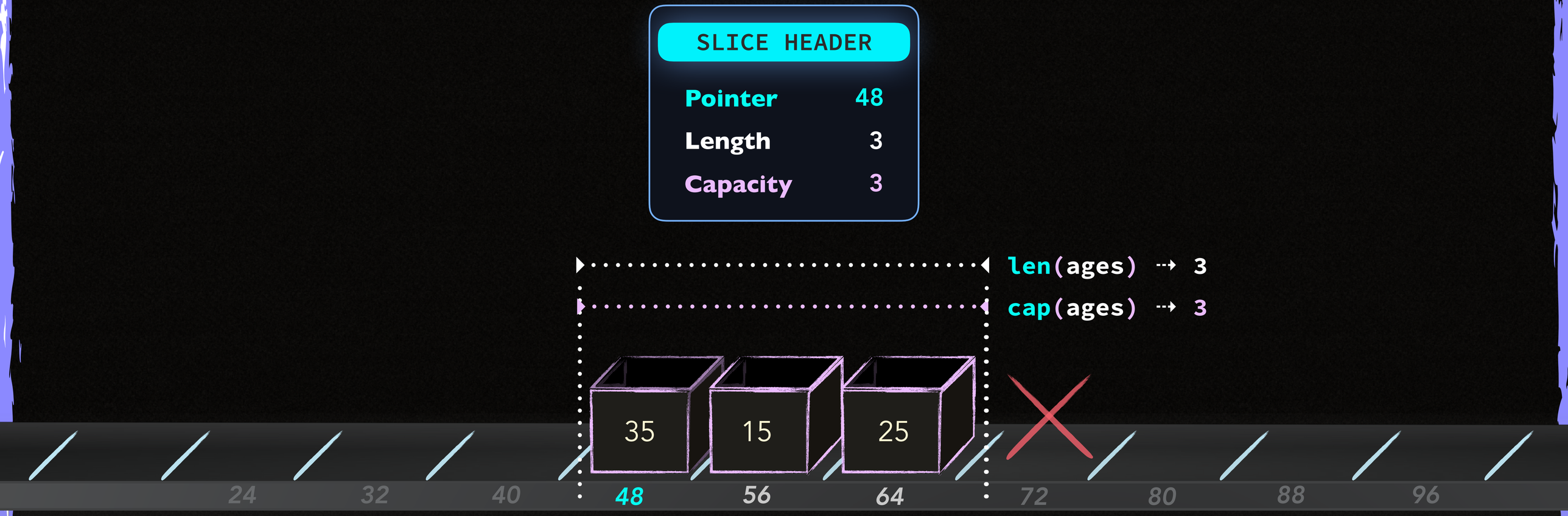
cap function returns the capacity of a slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

You cannot extend a slice beyond its capacity



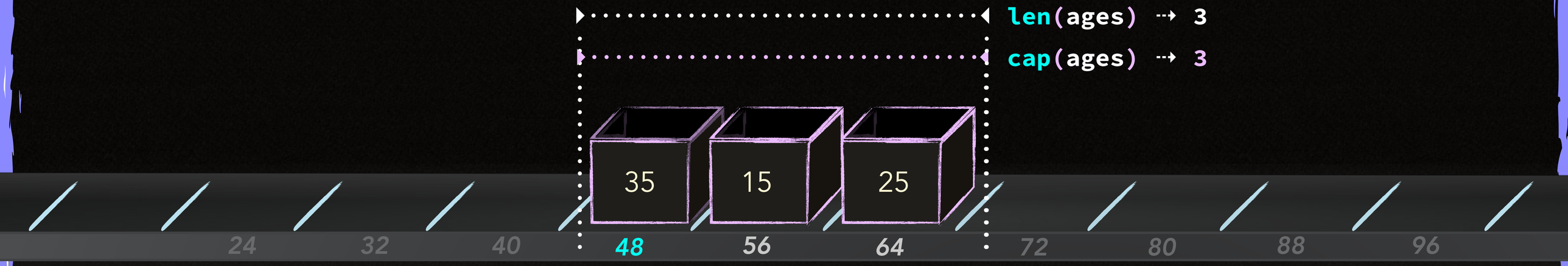

```
ages := []int{35, 15, 25}
ages = ages[0:0]
ages[0:cap(ages) + 1]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

Capacity is the length of the backing array after the slice's first item

SLICE HEADER	
Pointer	48
Length	3
Capacity	3

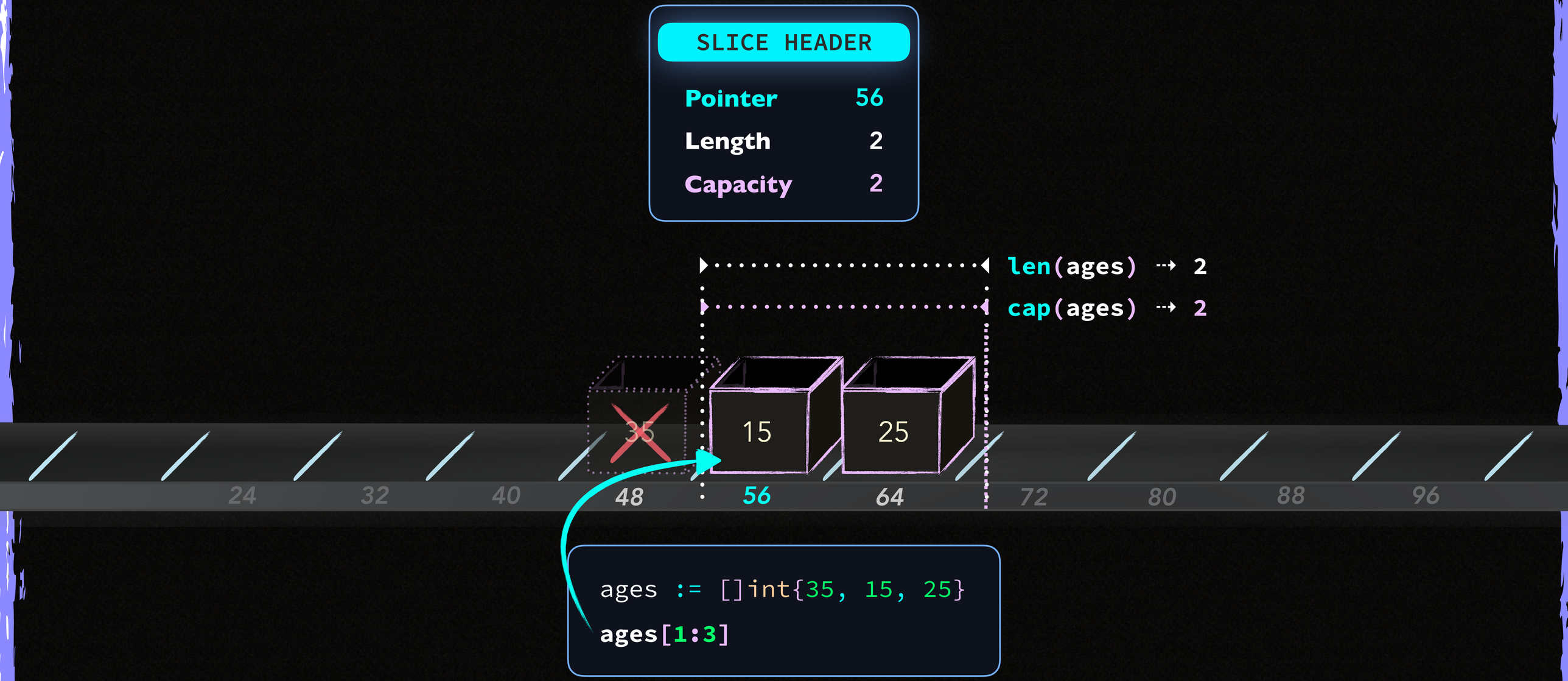



```
ages := []int{35, 15, 25}
ages[1:3]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

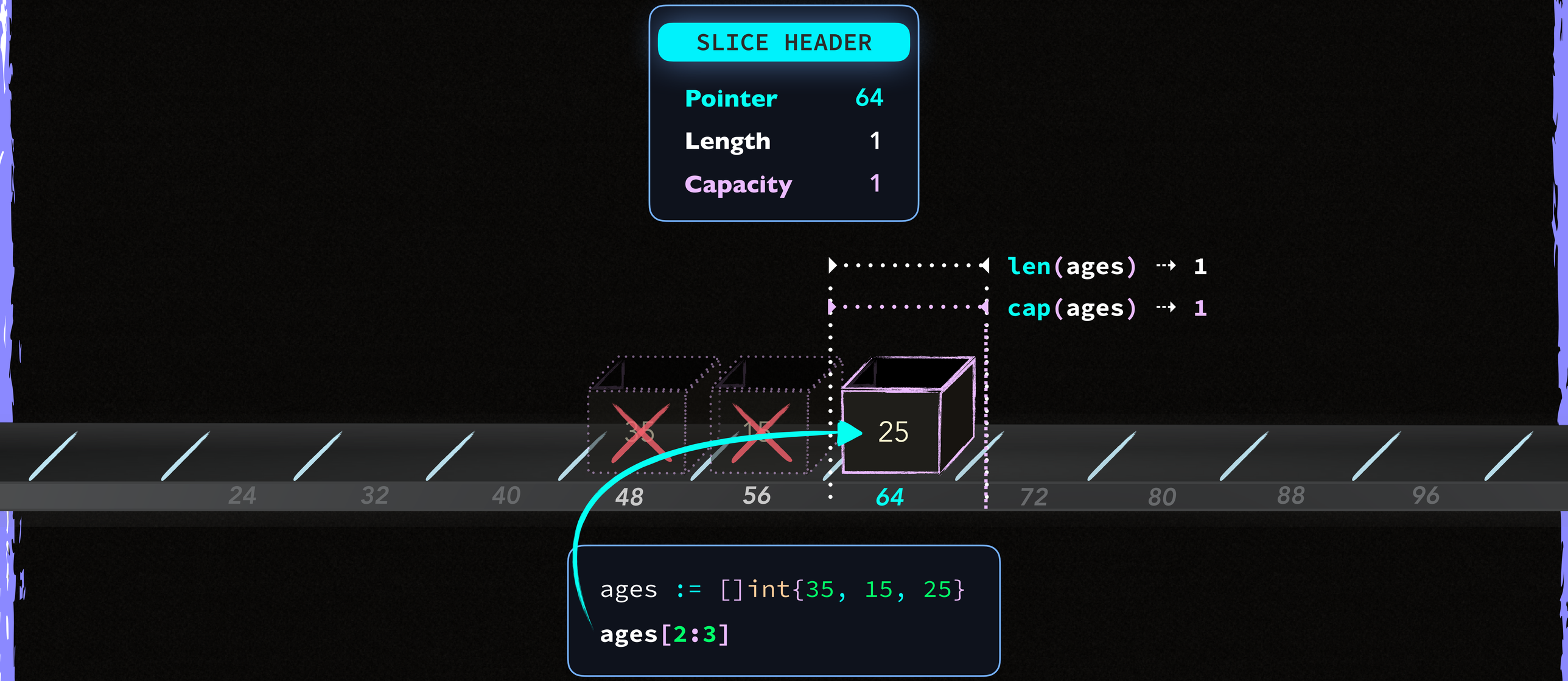
Capacity is the length of the backing array after the slice's first item



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

Capacity is the length of the backing array after the slice's first item

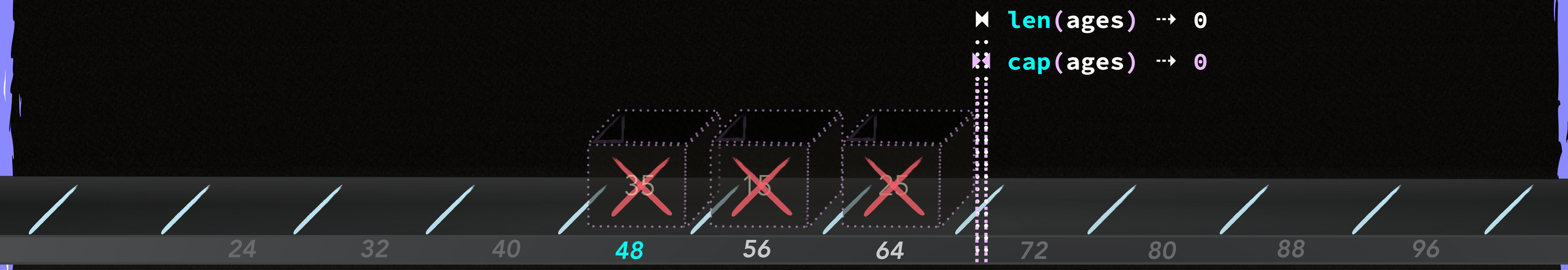


Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

An empty slice's length is zero but it has a backing array

SLICE HEADER	
Pointer	48
Length	0
Capacity	0



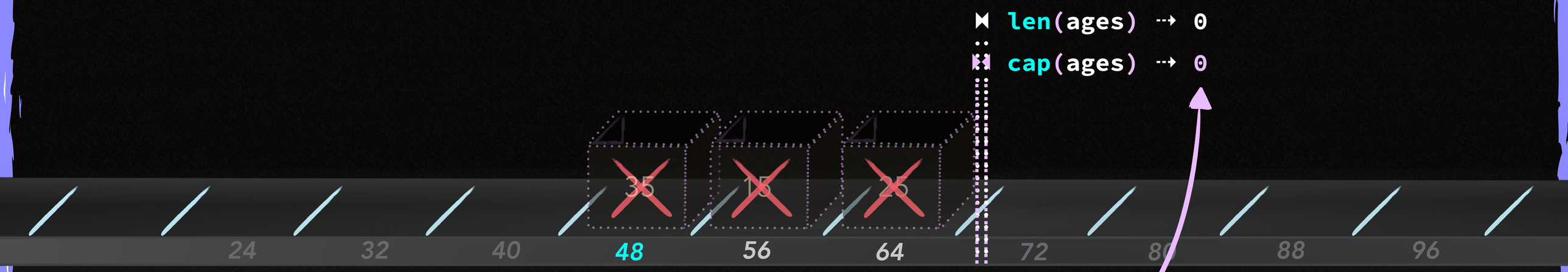

```
ages := []int{35, 15, 25}
ages[3:3]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

You cannot extend back when you lost the original slice header

SLICE HEADER	
Pointer	48
Length	0
Capacity	0




```
ages := []int{35, 15, 25}
ages := ages[3:3]
ages[0:3]
```

Copyrigh ncgumus

SLICE INTERNALS

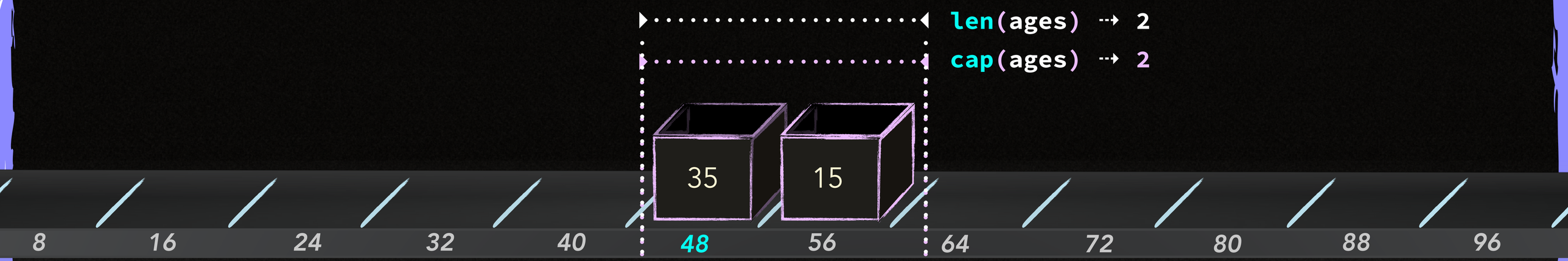
THE MECHANICS OF APPEND

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

What would happen if you append one more element to this slice?

```
ages := []int{35, 15}
```



SLICE HEADER	
Pointer	48
Length	2
Capacity	2

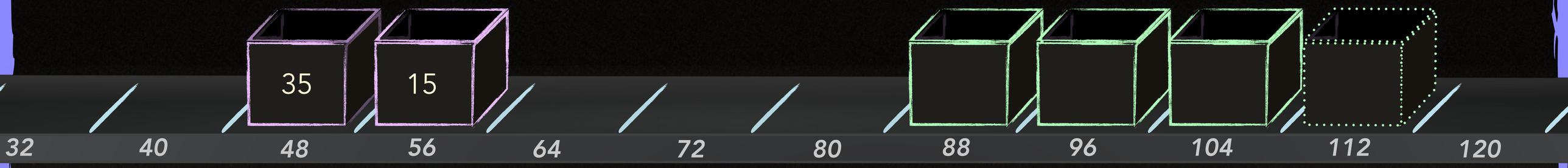
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

When the capacity is full; append() allocates a new and a larger array

```
ages := []int{35, 15}
```

```
ages = append(ages, 5)
```



SLICE HEADER	
Pointer	48
Length	2
Capacity	2

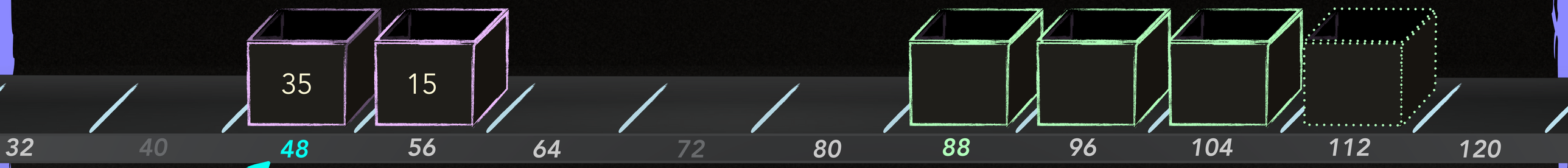
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

Returns a new slice header that points to the newly allocated array

```
ages := []int{35, 15}
```

```
ages = append(ages, 5)
```



SLICE HEADER	
Pointer	48
Length	2
Capacity	2

SLICE HEADER	
Pointer	88
Length	3
Capacity	4

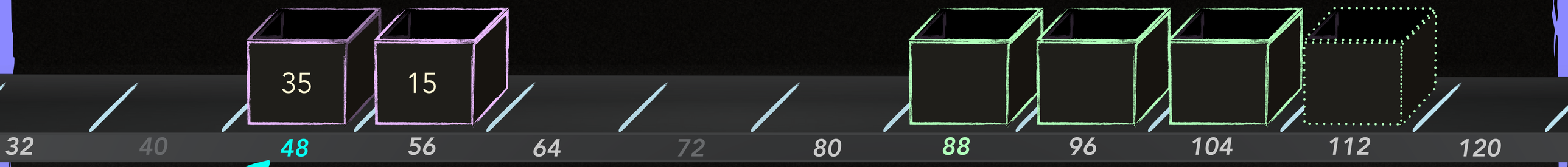
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

Why append() allocates a larger array?

```
ages := []int{35, 15}
```

```
ages = append(ages, 5)
```



SLICE HEADER	
Pointer	48
Length	2
Capacity	2

SLICE HEADER	
Pointer	88
Length	3
Capacity	4

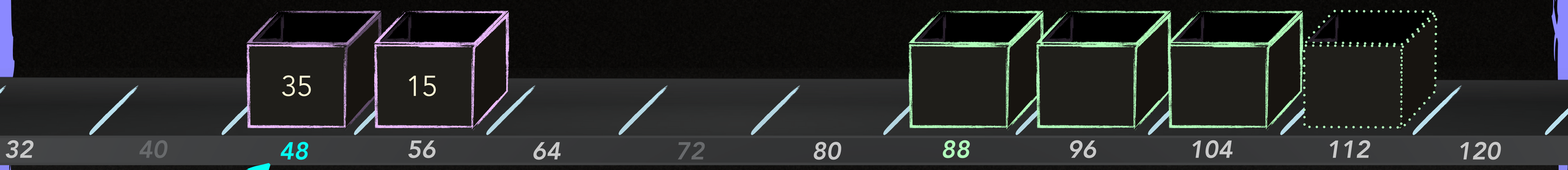
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

Allocating a larger array reduces the number of allocations in the future

```
ages := []int{35, 15}
```

```
ages = append(ages, 5)
```



SLICE HEADER	
Pointer	48
Length	2
Capacity	2

SLICE HEADER	
Pointer	88
Length	3
Capacity	4

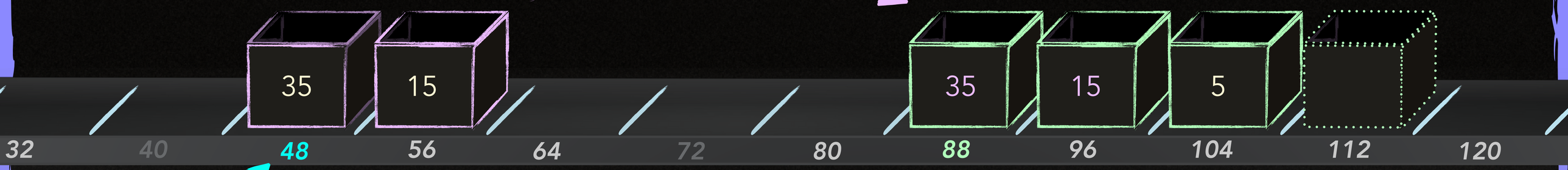
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

Copies the elements of the previous array to the new array

```
ages := []int{35, 15}
```

```
ages = append(ages, 5)
```



SLICE HEADER	
Pointer	48
Length	2
Capacity	2

SLICE HEADER	
Pointer	88
Length	3
Capacity	4

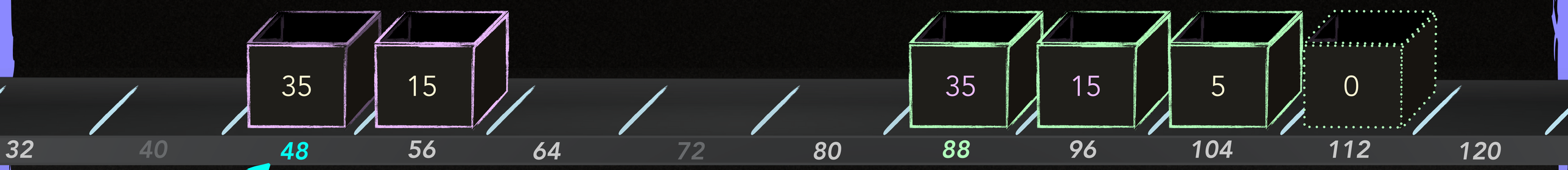
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

Assigns zero values to uninitialized elements of the backing array

```
ages := []int{35, 15}
```

```
ages = append(ages, 5)
```



SLICE HEADER	
Pointer	48
Length	2
Capacity	2

SLICE HEADER	
Pointer	88
Length	3
Capacity	4

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

FULL SLICE EXPRESSIONS

Sometimes you need to " limit " yourself

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

```
newSlice := slicable[START:STOP]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

```
newSlice := slicable[START:STOP:CAP]
```

"CAP" limits the returned slice's capacity

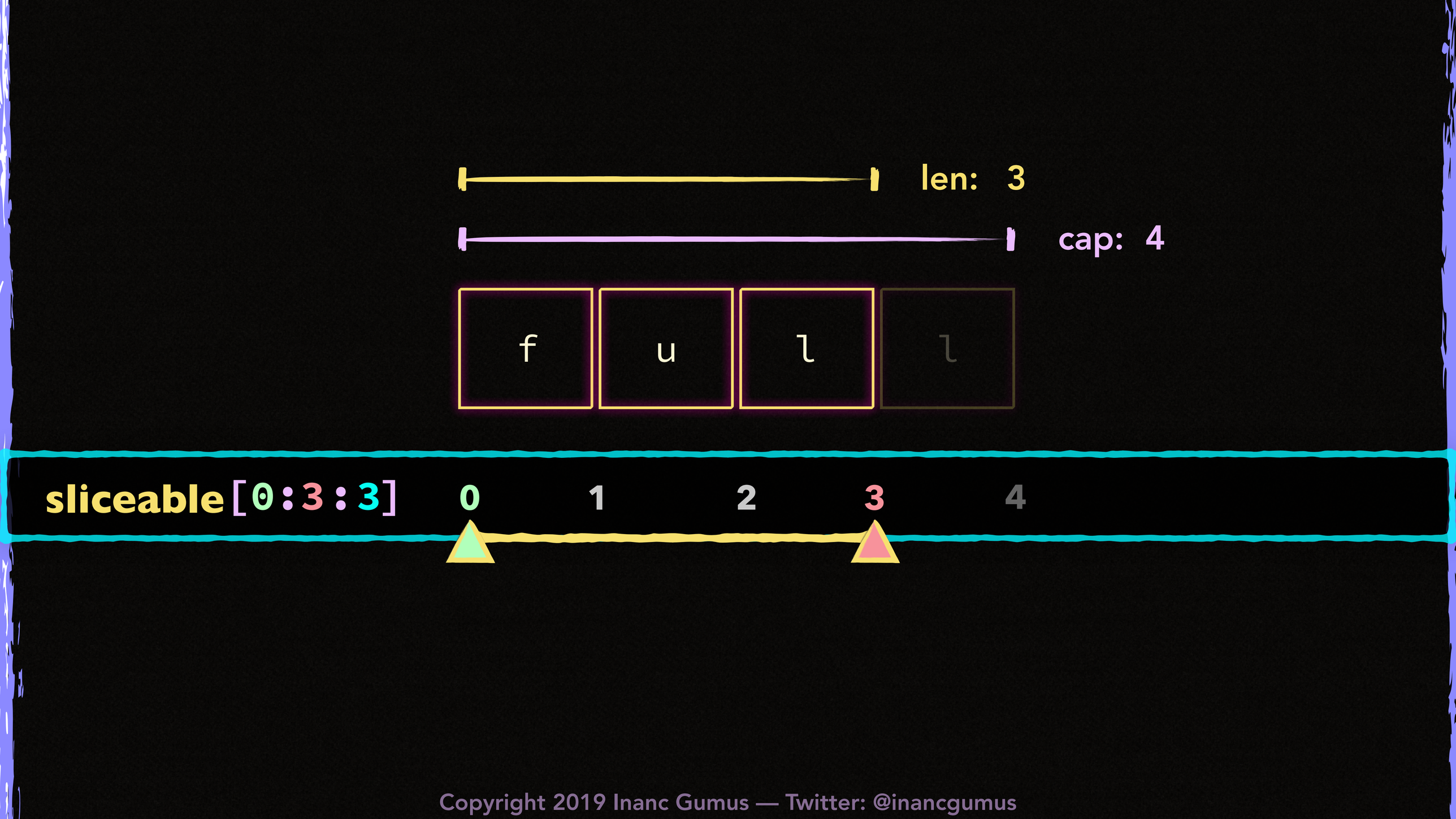
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

```
sliceable := []byte{'f', 'u', 'l', 'l'}
```

sliceable[0:3] 0 1 2 3 4

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

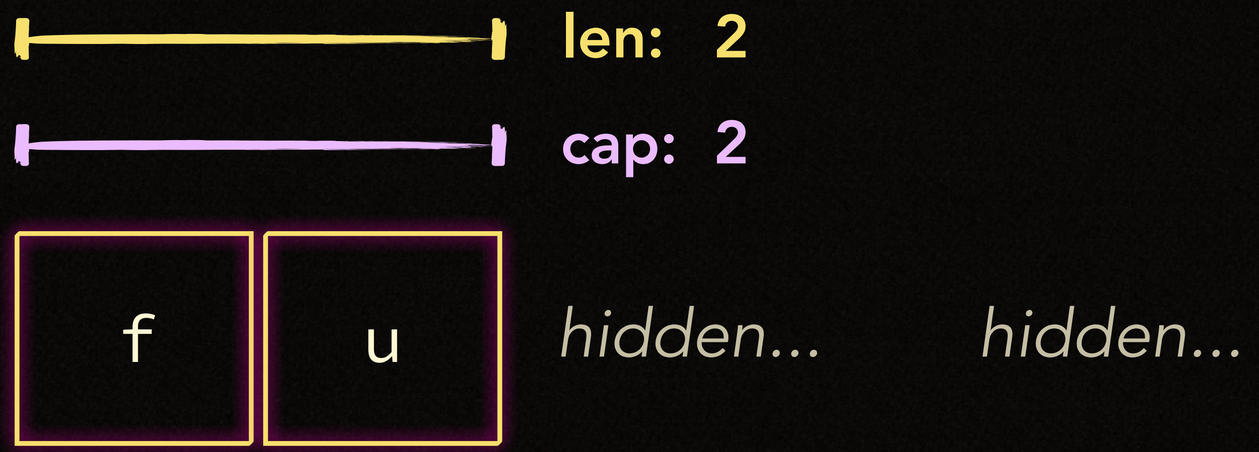
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

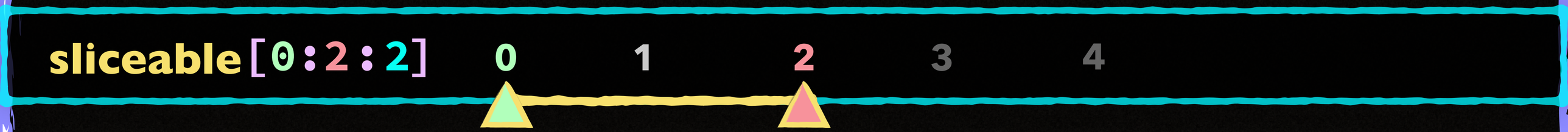


Copyright 2019 Inanc Gumus — Twitter: @inancgumus

think of the **capacity position** as
the element position in **the backing array**

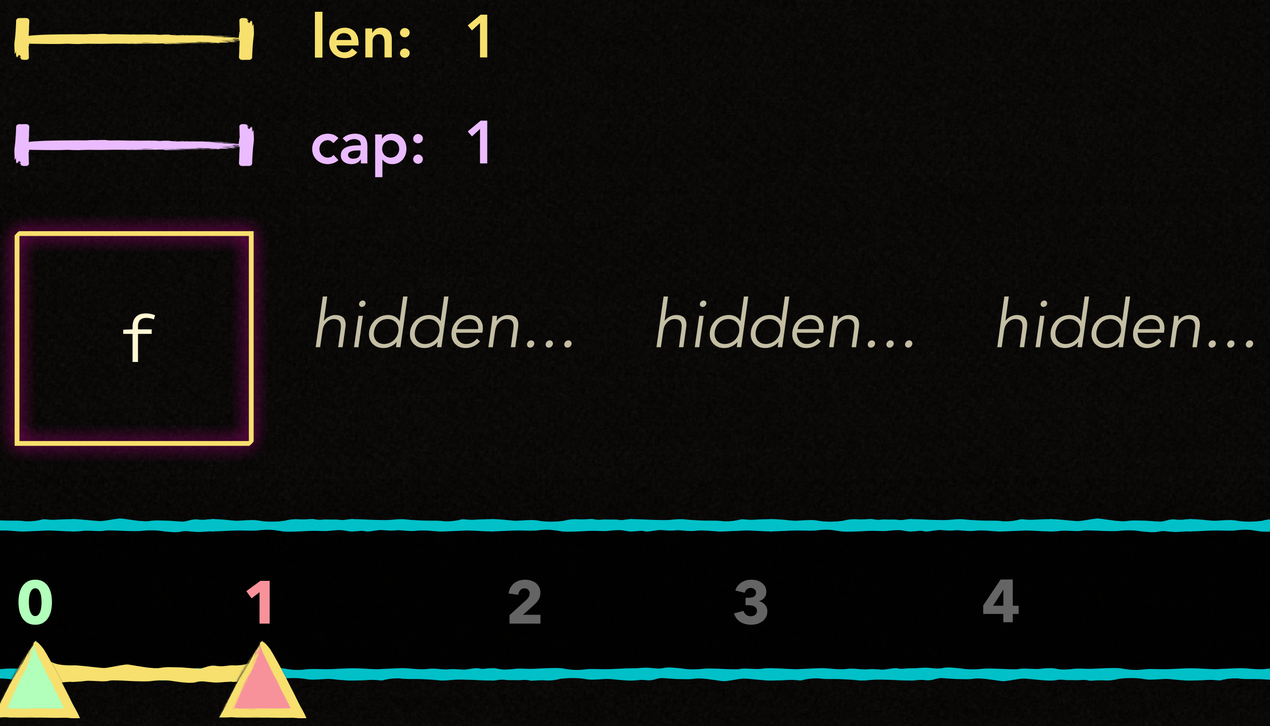
Copyright 2019 Inanc Gumus — Twitter: @inancgumus



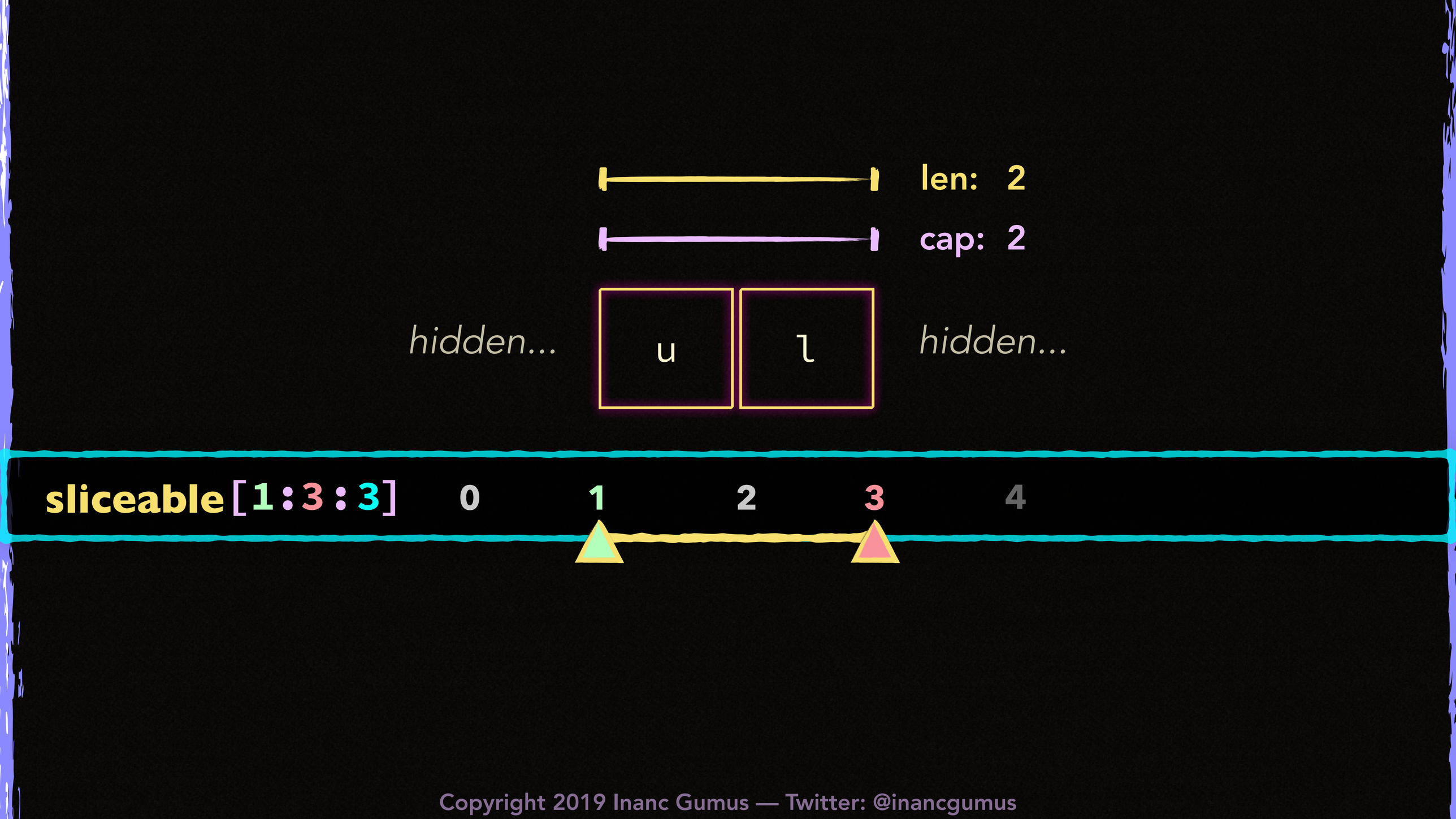


stop position <= capacity position

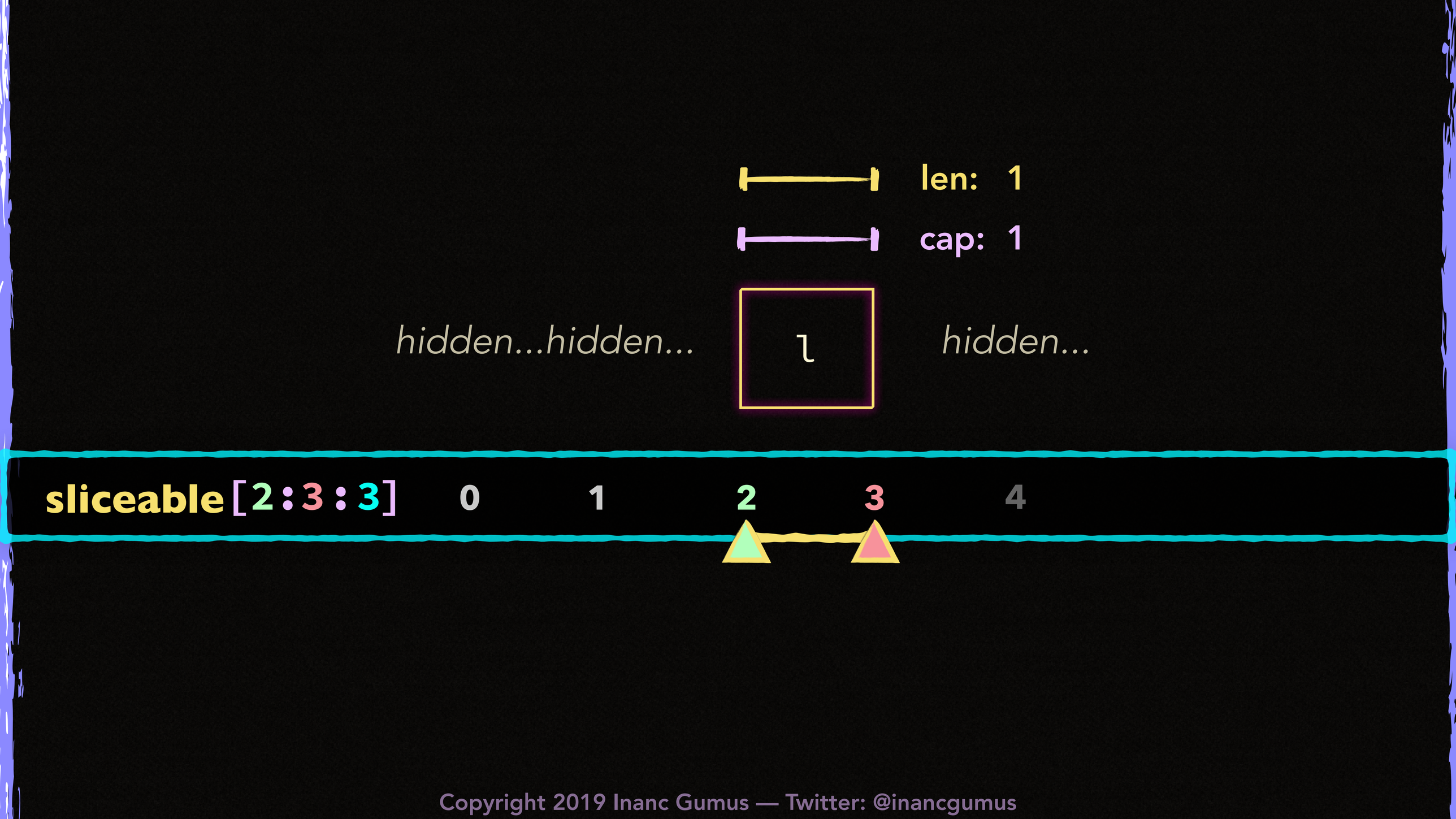
Copyright 2019 Inanc Gumus — Twitter: @inancgumus



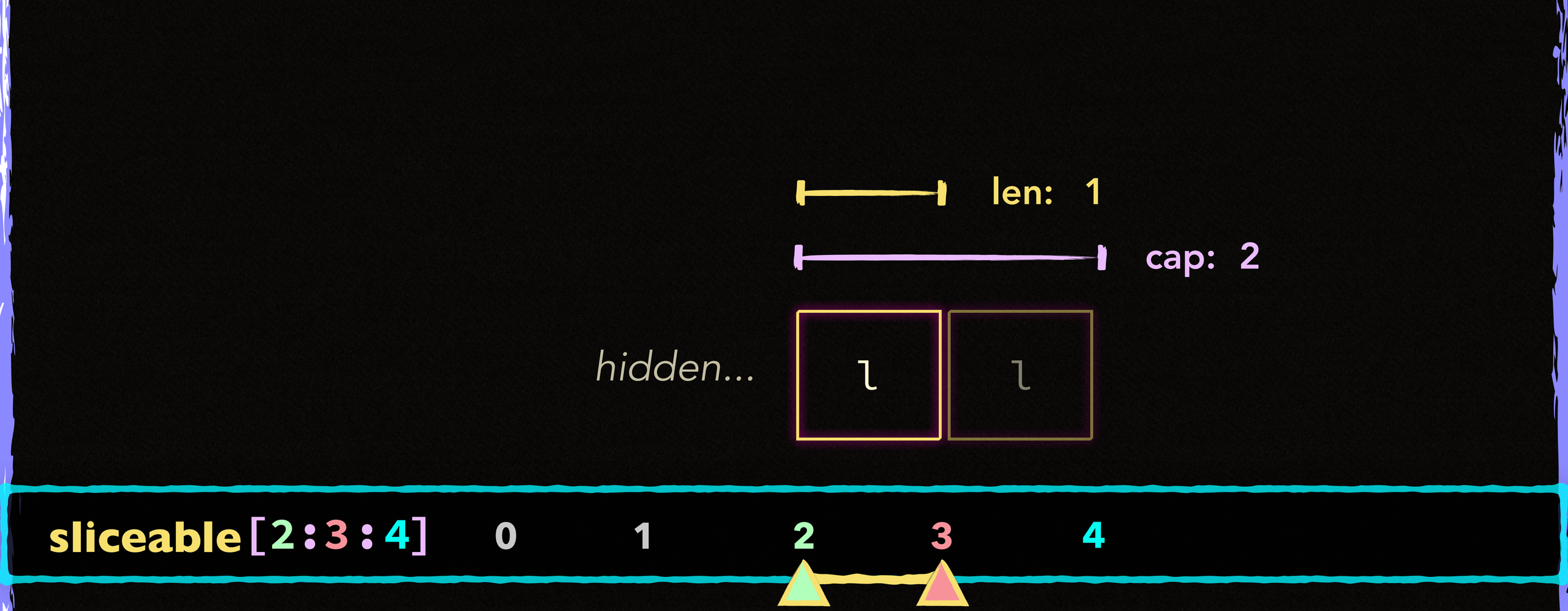
Copyright 2019 Inanc Gumus — Twitter: @inancgumus



Copyright 2019 Inanc Gumus — Twitter: @inancgumus



Copyright 2019 Inanc Gumus — Twitter: @inancgumus



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

len: 0

cap: 0

hidden... *hidden...* *hidden...* *hidden...*

sliceable[4:4:4] 0 1 2 3 4

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

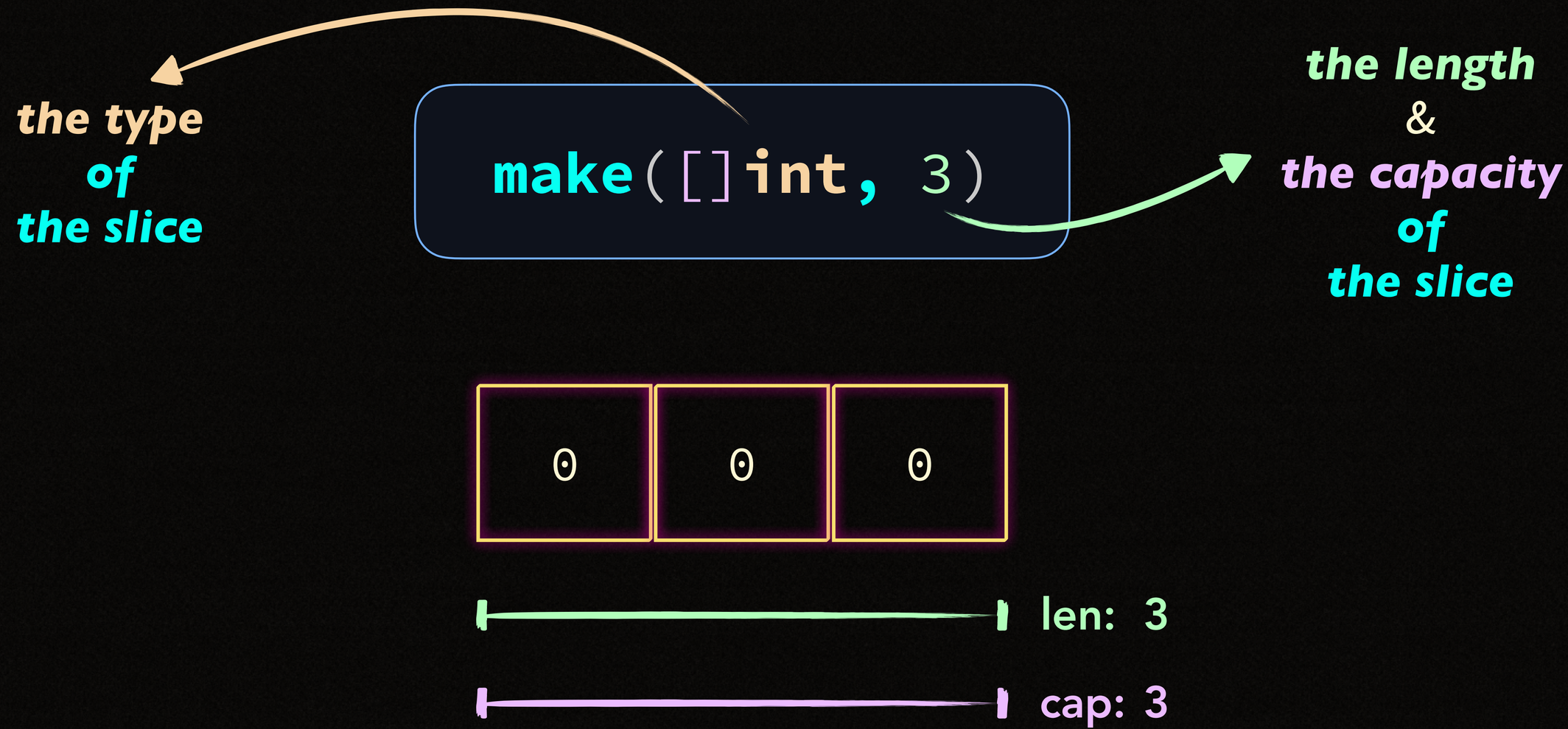
MAKE

Preallocate Backing Arrays

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MAKE

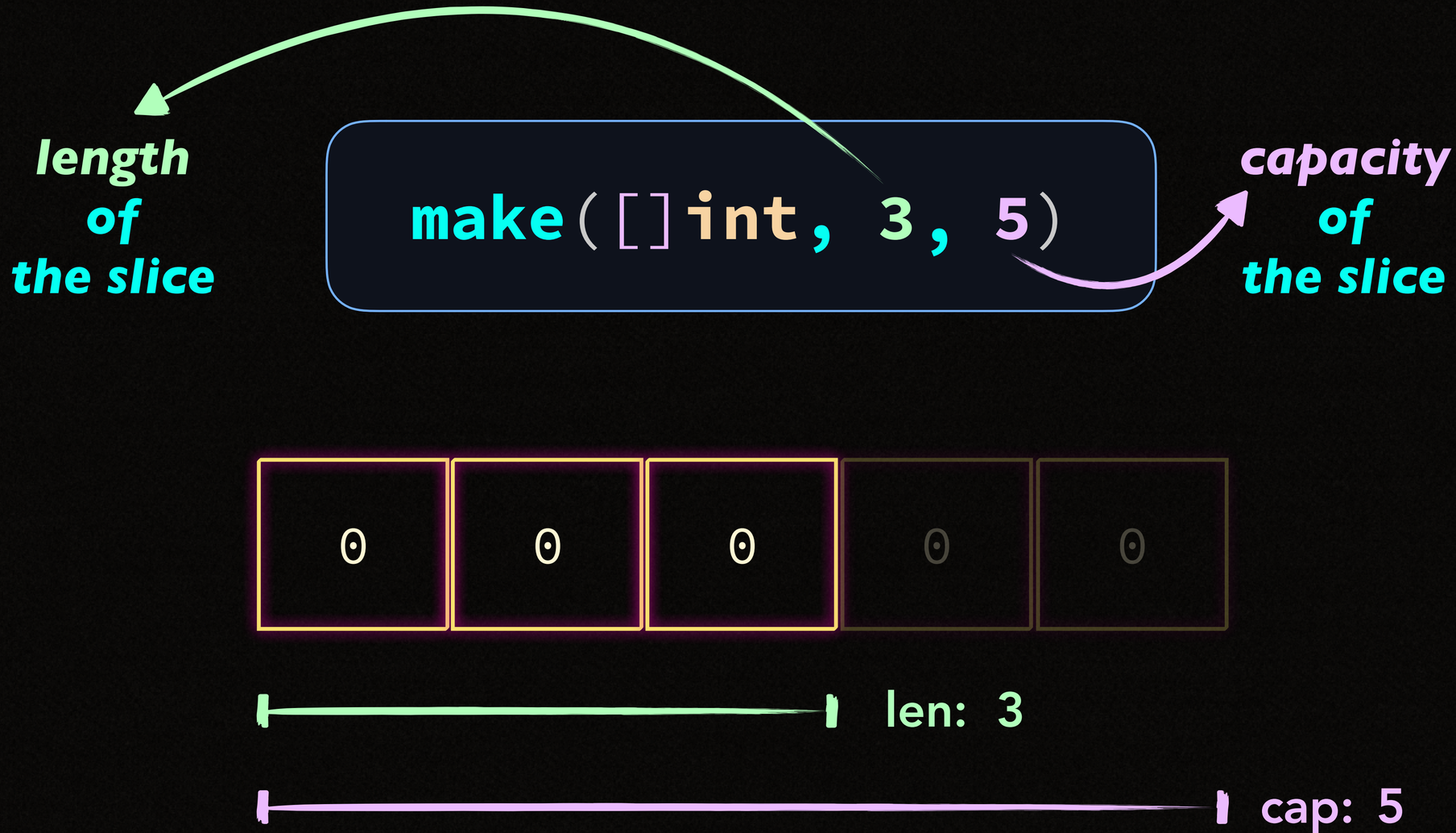
make initializes and returns a slice with the given length and capacity



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MAKE

make initializes and returns a slice with the given length and capacity



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MAKE

when the given length is 0, the make() will return a slice, ready to append

```
make([]int, 0, 5)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MAKE

when the given length is 0, the make() will return a slice, ready to append

ready to append

```
s := make([]int, 0, 5)
s  = append(s, 42)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MAKE

when the given length is 0, the make() will return a slice, ready to append

```
s := make([]int, 0, 5)
s  = append(s, 42)
```

ready to append

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

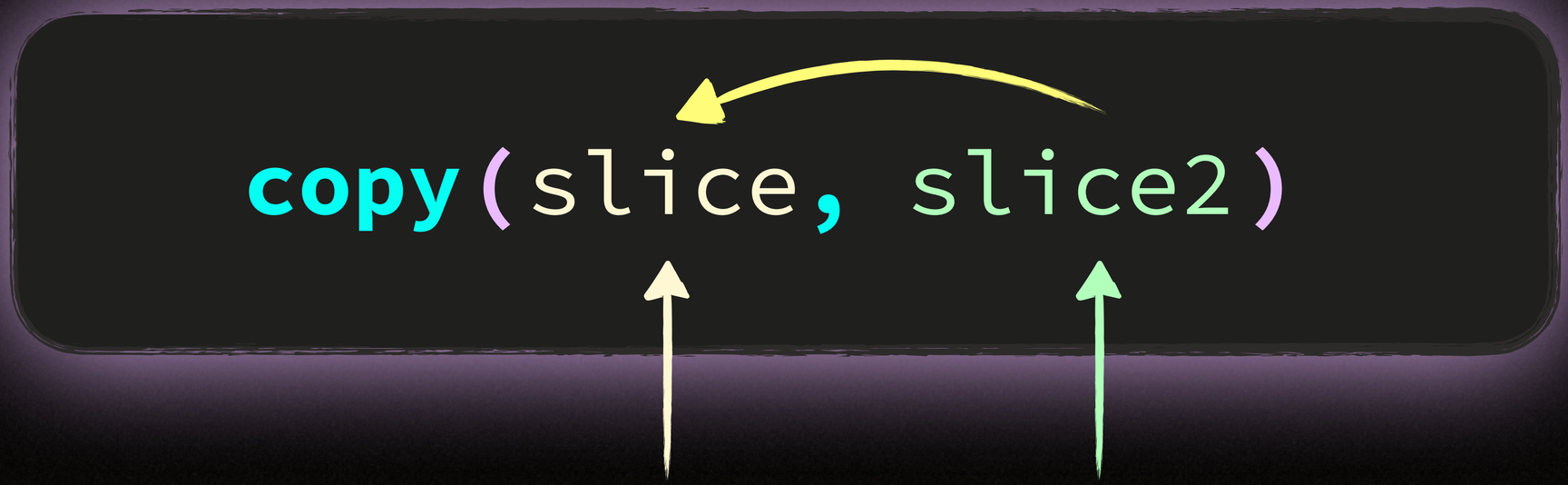
COPY

Copies the elements of a slice to another slice

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

COPY

Copies the elements of a slice to another slice



the element types of the slices should be identical

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

COPY

Copies the elements of a slice to another slice

```
evens := []int{2, 4}
```

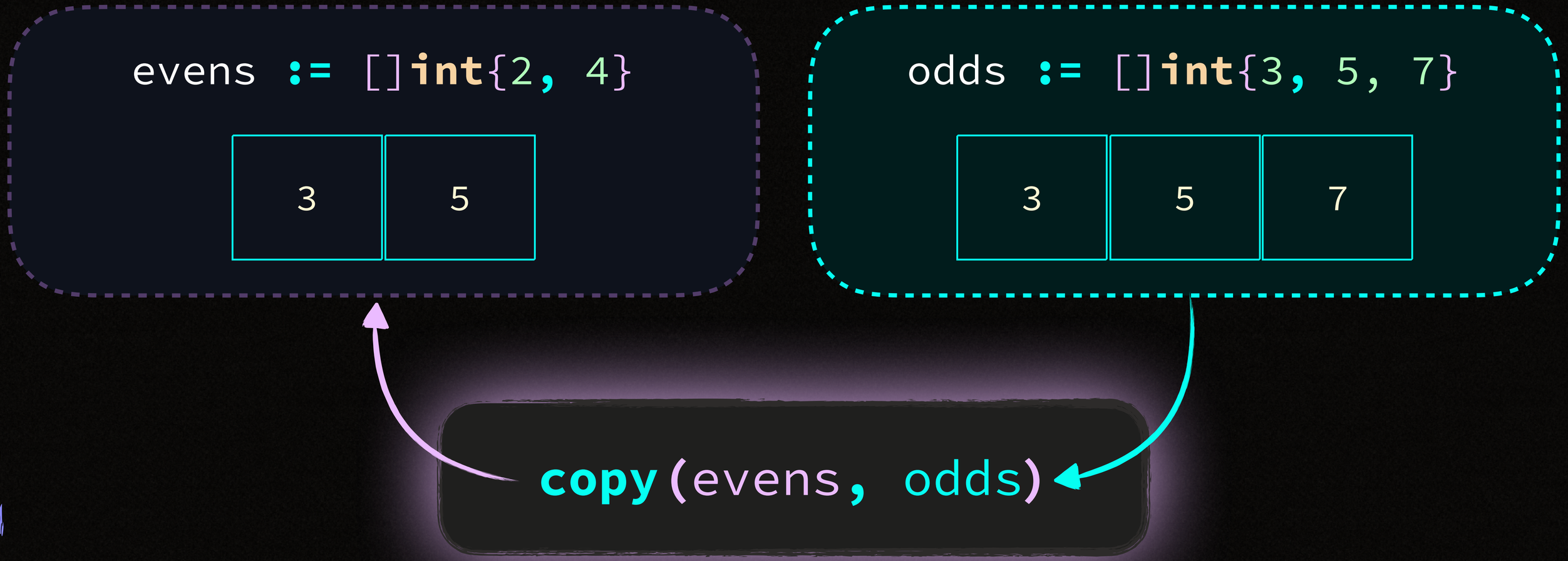
```
odds := []int{3, 5, 7}
```

```
copy(evens, odds)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

COPY

Copies the elements *based on* the length of the smallest slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

COPY

It returns the number of copied elements

```
evens := []int{2, 4}
```

```
odds := []int{3, 5, 7}
```

```
N := copy(evens, odds)
```

2 elements are copied

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

summary

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICES vs ARRAYS

Slices are dynamic — Arrays are fixed

SLICE

```
var nums []int
```

Its length is *not* part of its type

It can grow and shrink

Its zero value is nil

ARRAY

```
var nums [2]int
```

Its length is part of its type

It cannot grow or shrink

Its zero value is zero valued elements

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

You can only compare a slice value to a nil value

```
var nums []int    ==    nil
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

nil slice

```
var nums []int
```

empty slice

```
nums := []int{}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

Appends an element to a slice and returns a new slice

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

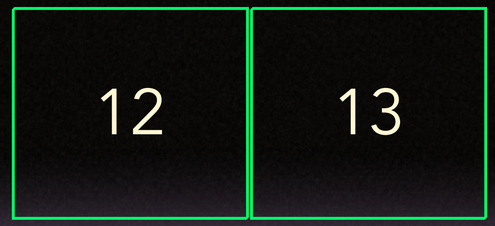
It appends the elements after the length of the given slice

```
nums := []int{1, 2, 3}
nums = append(nums, 4, 9)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

APPEND

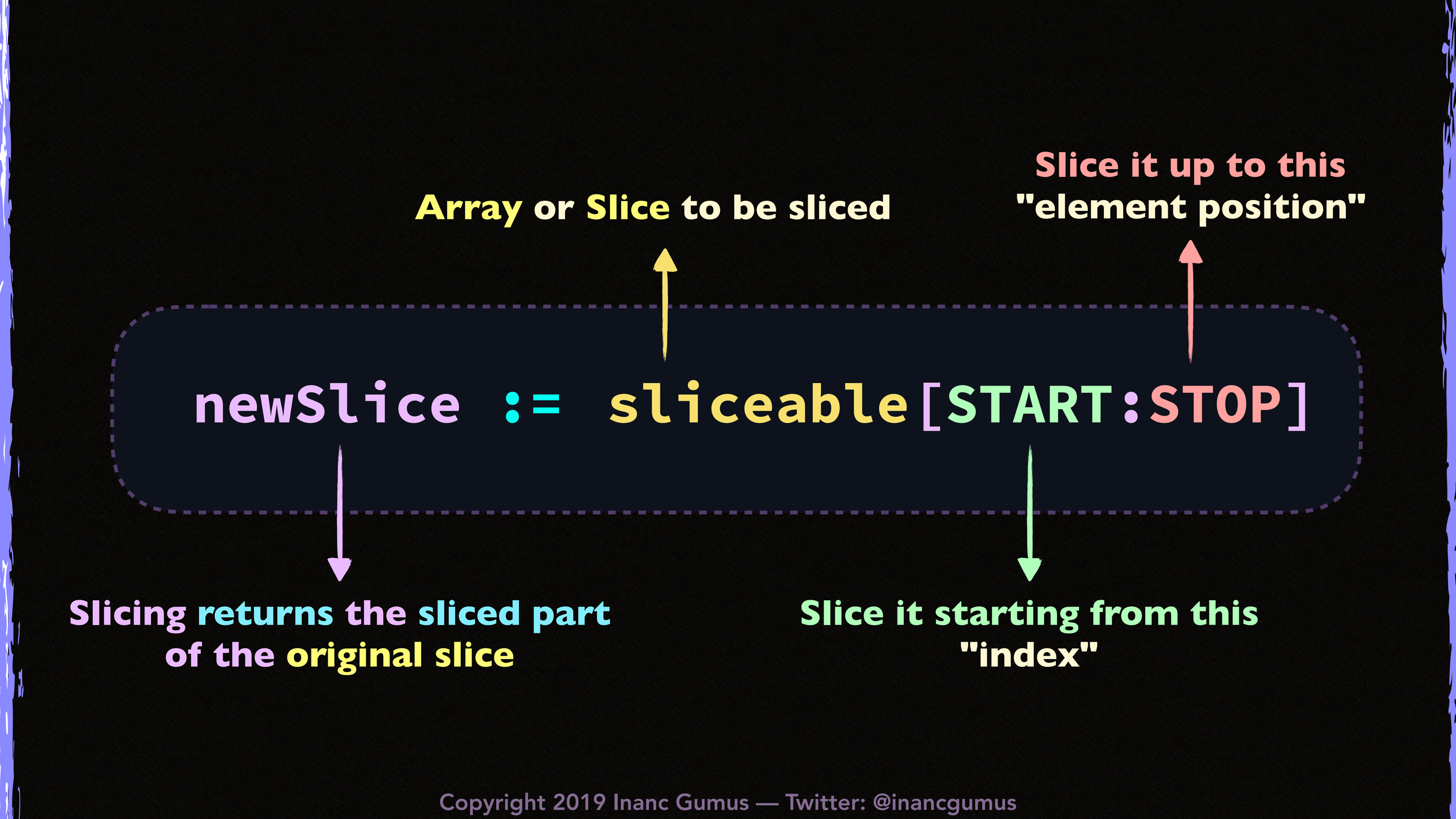
Ellipsis allows you to append a slice to another slice




```
nums := []int{1, 2, 3}
tens := []int{12, 13}
nums = append(nums, tens...)
```

```
append(nums, 12, 13)
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE EXPRESSIONS

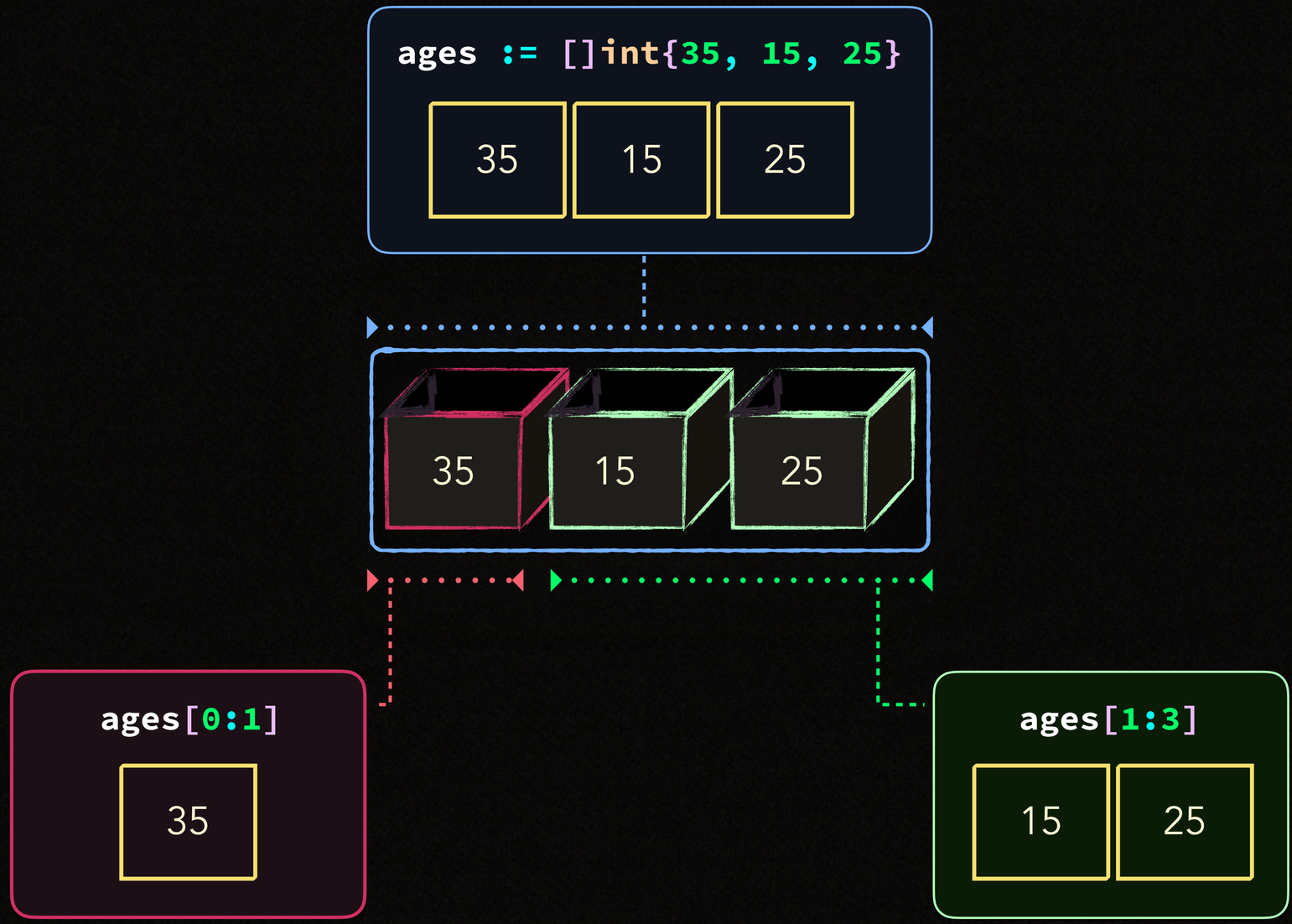
Creates a new slice from 1st index to the 4th element

```
msg := []byte{'h', 'e', 'l', 'l', 'o'}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

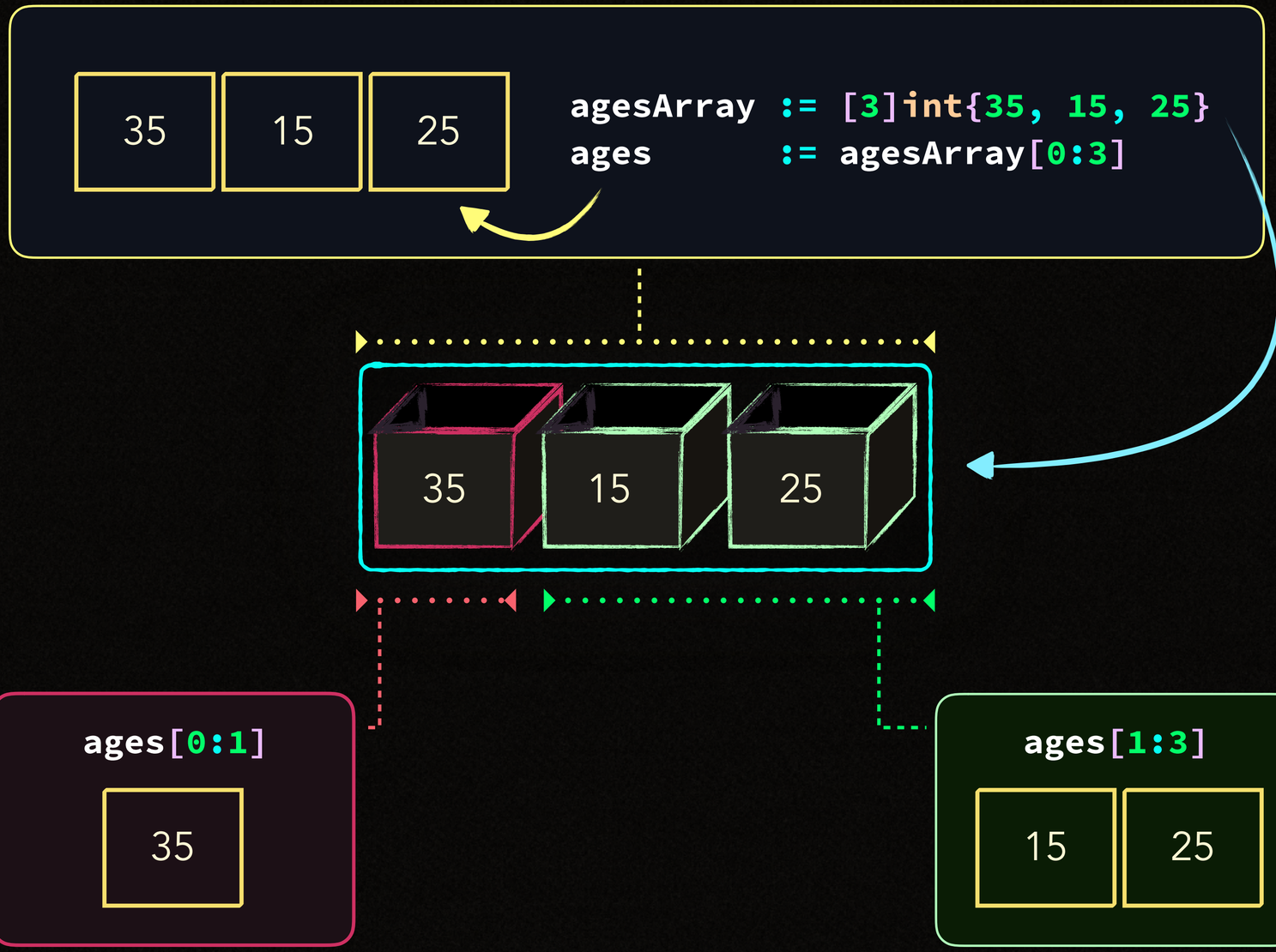
A slice is a window to its backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

BACKING ARRAY

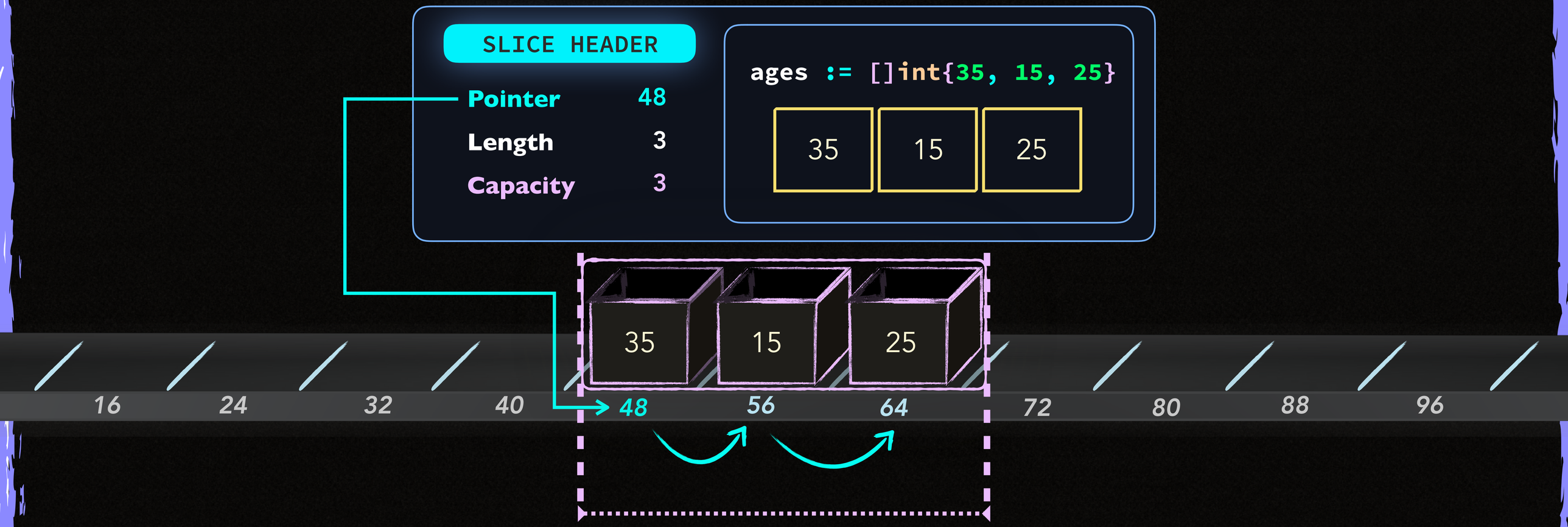
A sliced array becomes the backing array of the returned slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

SLICE HEADER

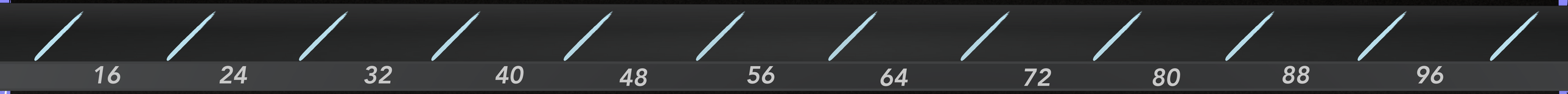
A tiny data structure that refers to a slice's backing array



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

NIL SLICE

An nil slice doesn't have a backing array but it has a slice header



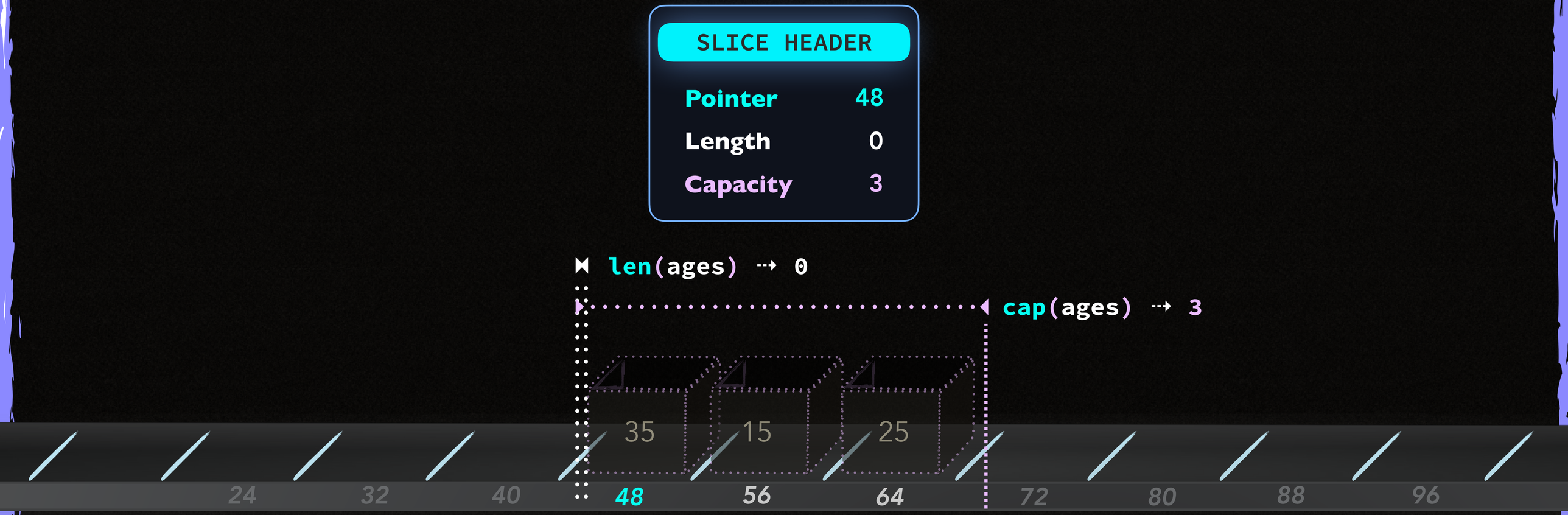
When you don't know whether a **slice** will ever have any elements

declare it as a nil slice

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

An empty slice's length is zero but it has a backing array

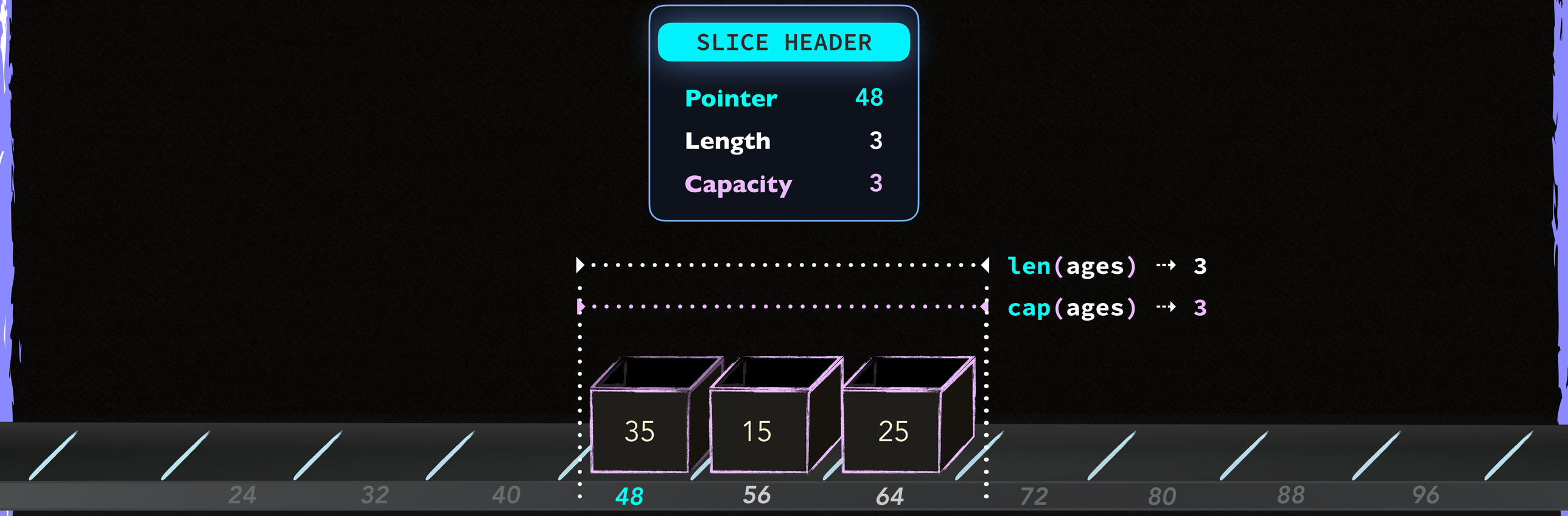



```
ages := []int{35, 15, 25}
ages[0:0]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

Capacity determines how much you can extend a slice

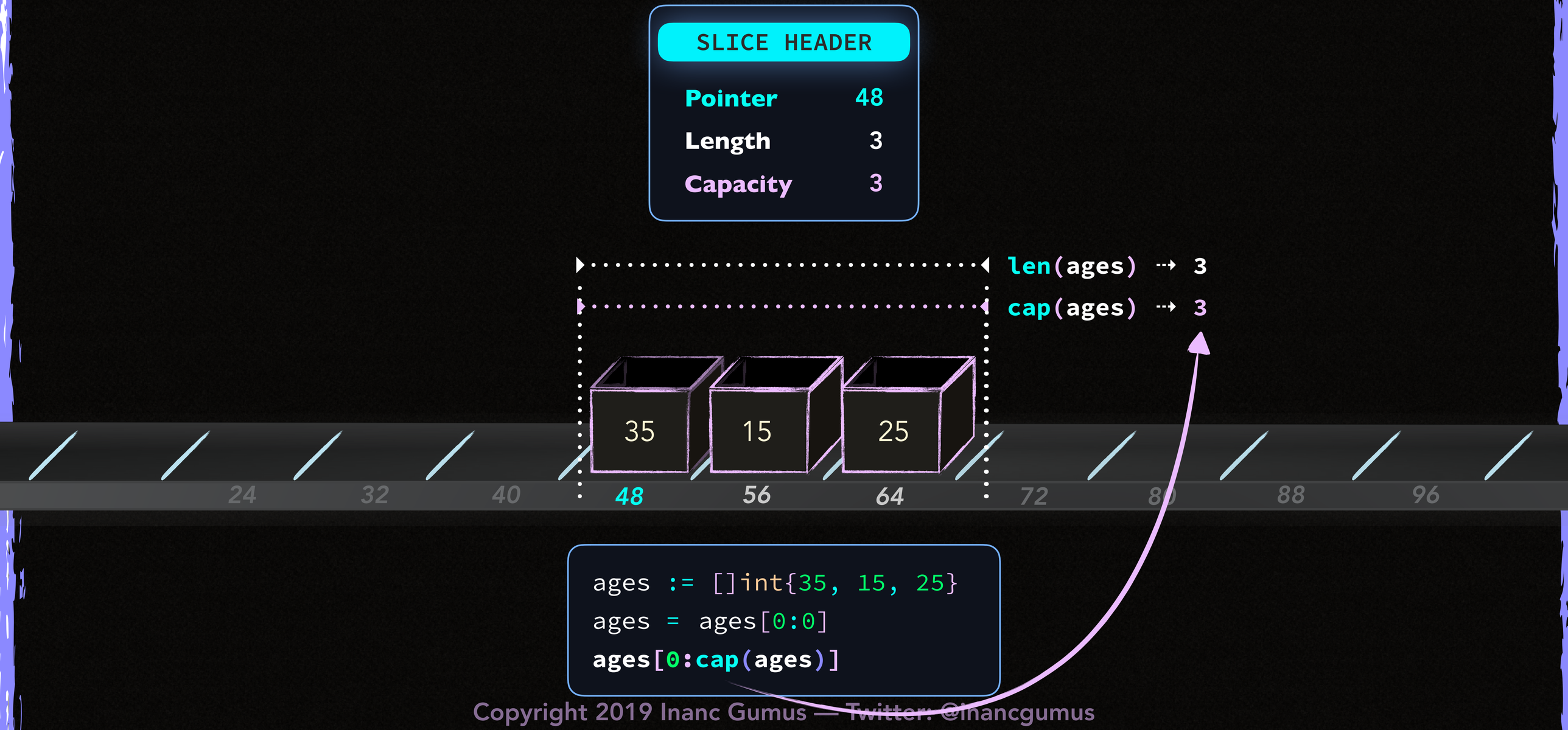



```
ages := []int{35, 15, 25}
ages = ages[0:0]
ages[0:3]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

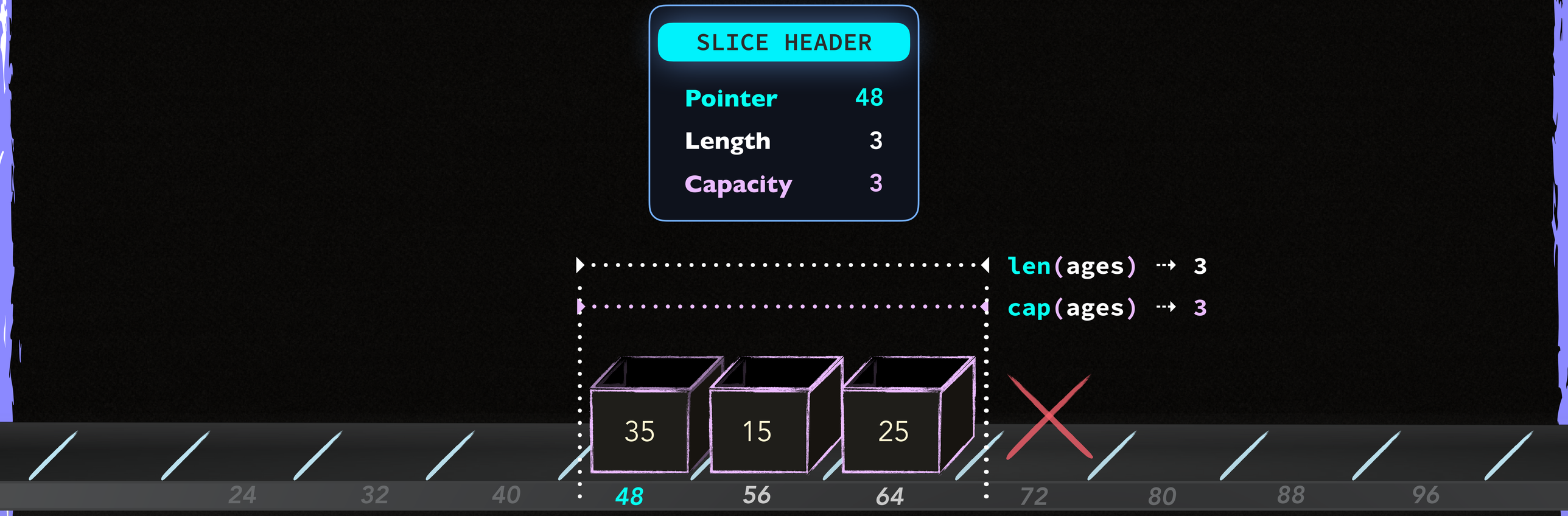
cap function returns the capacity of a slice



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

CAPACITY

You cannot extend a slice beyond its capacity




```
ages := []int{35, 15, 25}
ages = ages[0:0]
ages[0:cap(ages) + 1]
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

Append copies the elements of the previous array to the new array

```
ages := []int{35, 15}
```

```
ages = append(ages, 5)
```

SLICE HEADER	
Pointer	48
Length	2
Capacity	2

SLICE HEADER	
Pointer	88
Length	3
Capacity	4

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MECHANICS OF APPEND

Assigns zero values to uninitialized elements of the backing array

```
ages := []int{35, 15}
```

```
ages = append(ages, 5)
```

SLICE HEADER	
Pointer	48
Length	2
Capacity	2

SLICE HEADER	
Pointer	88
Length	3
Capacity	4

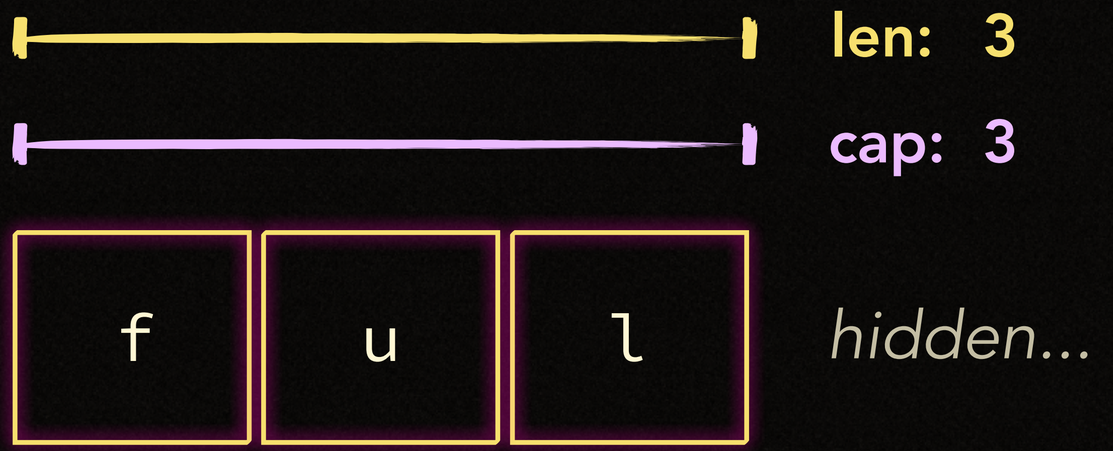
Copyright 2019 Inanc Gumus — Twitter: @inancgumus

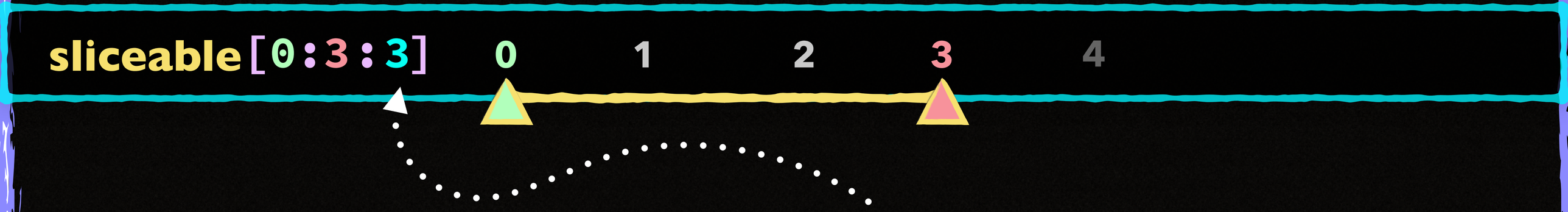
FULL SLICE EXPRESSIONS

```
sliceable := []byte{'f', 'u', 'l', 'l'}
```

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

FULL SLICE EXPRESSIONS



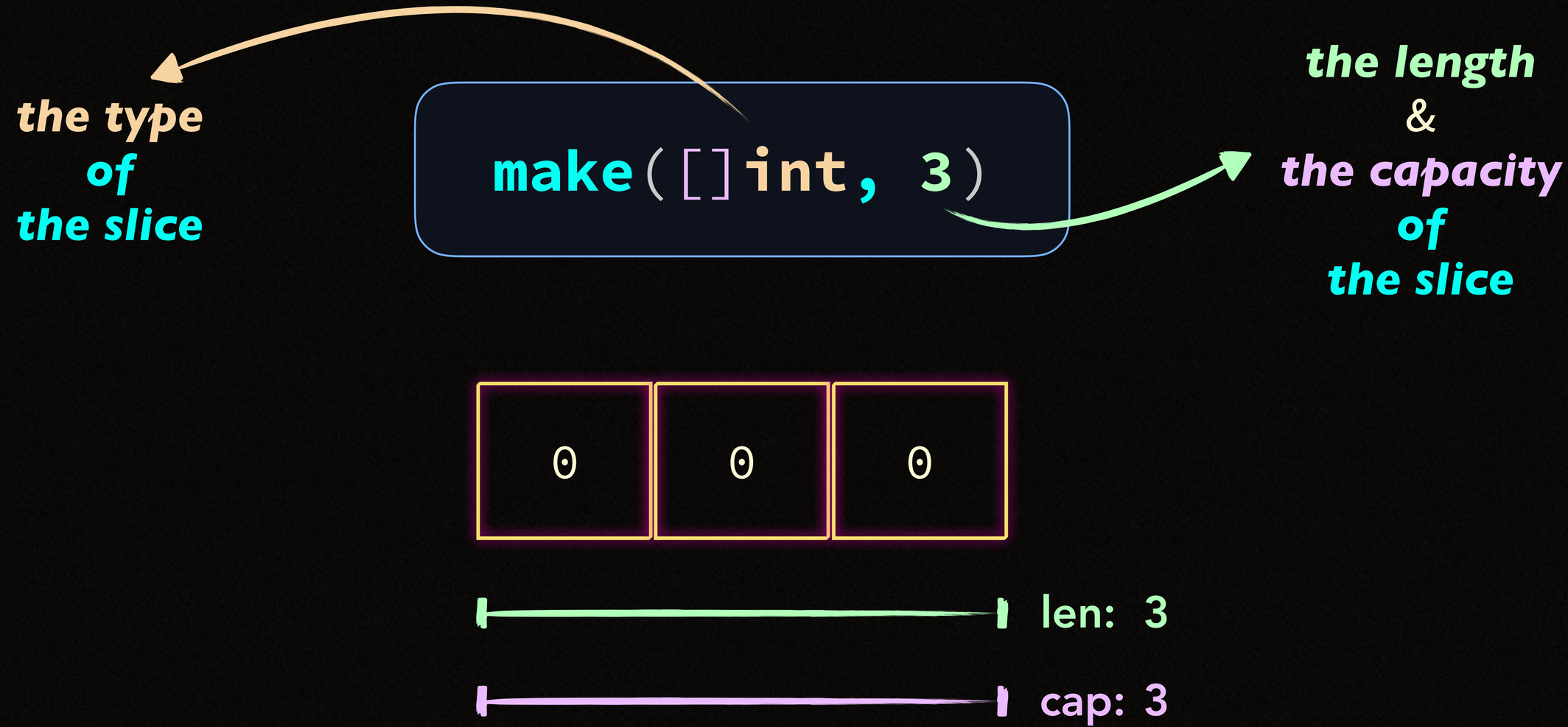


think of the **capacity position** as
the element position in **the backing array**

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MAKE

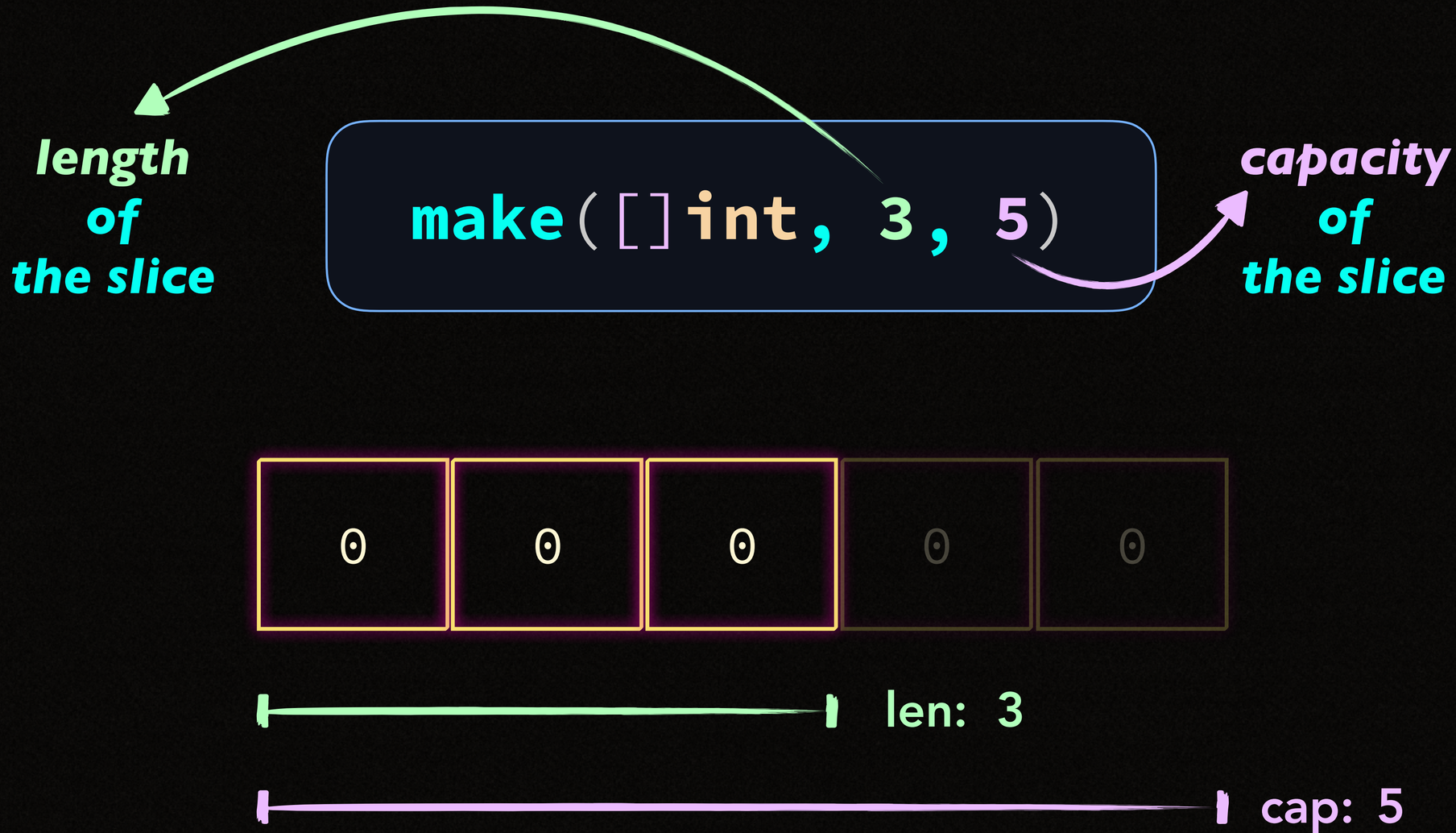
make initializes and returns a slice with the given length and capacity



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MAKE

make initializes and returns a slice with the given length and capacity



Copyright 2019 Inanc Gumus — Twitter: @inancgumus

MAKE

when the given length is 0, the make() will return a slice, ready to append

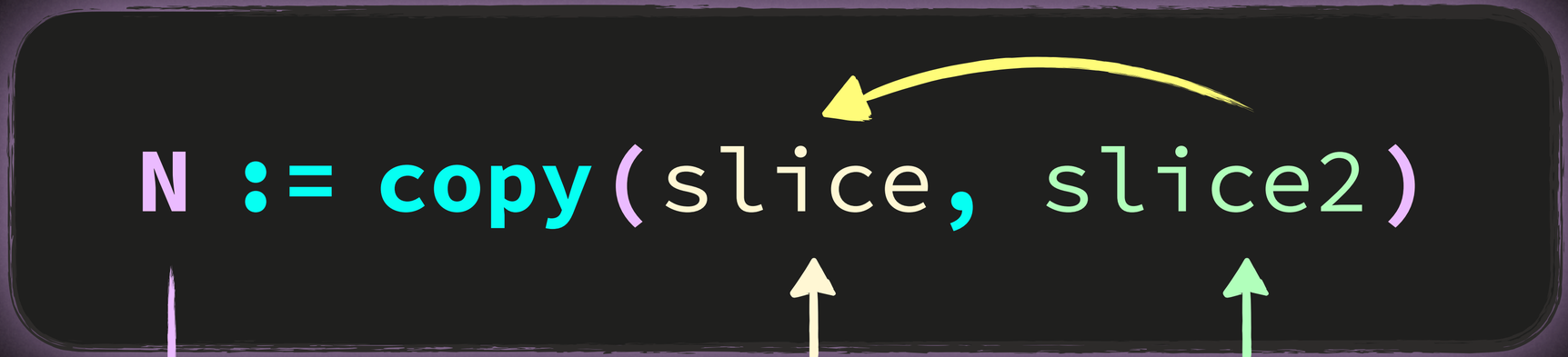
```
s := make([]int, 0, 5)
s  = append(s, 42)
```

ready to append

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

COPY

Copies the elements of a slice to another slice



returns the number of copied elements

destination slice

source slice

the element types of the slices should be identical

Copyright 2019 Inanc Gumus — Twitter: @inancgumus

PART IV

Composite Types

Arrays

Collection
of
Elements
Indexable
Fixed Length

Slices

Collection
of
Elements
Indexable
Dynamic length

String Internals

Byte Slices
ASCII & Unicode
Encoding & Decoding

Maps

Collection
of
Indexable
Key-Value Pairs

Structs

Groups
different types of
variables together

Copyright 2019 Inanc Gumus — Twitter: @inancgumus